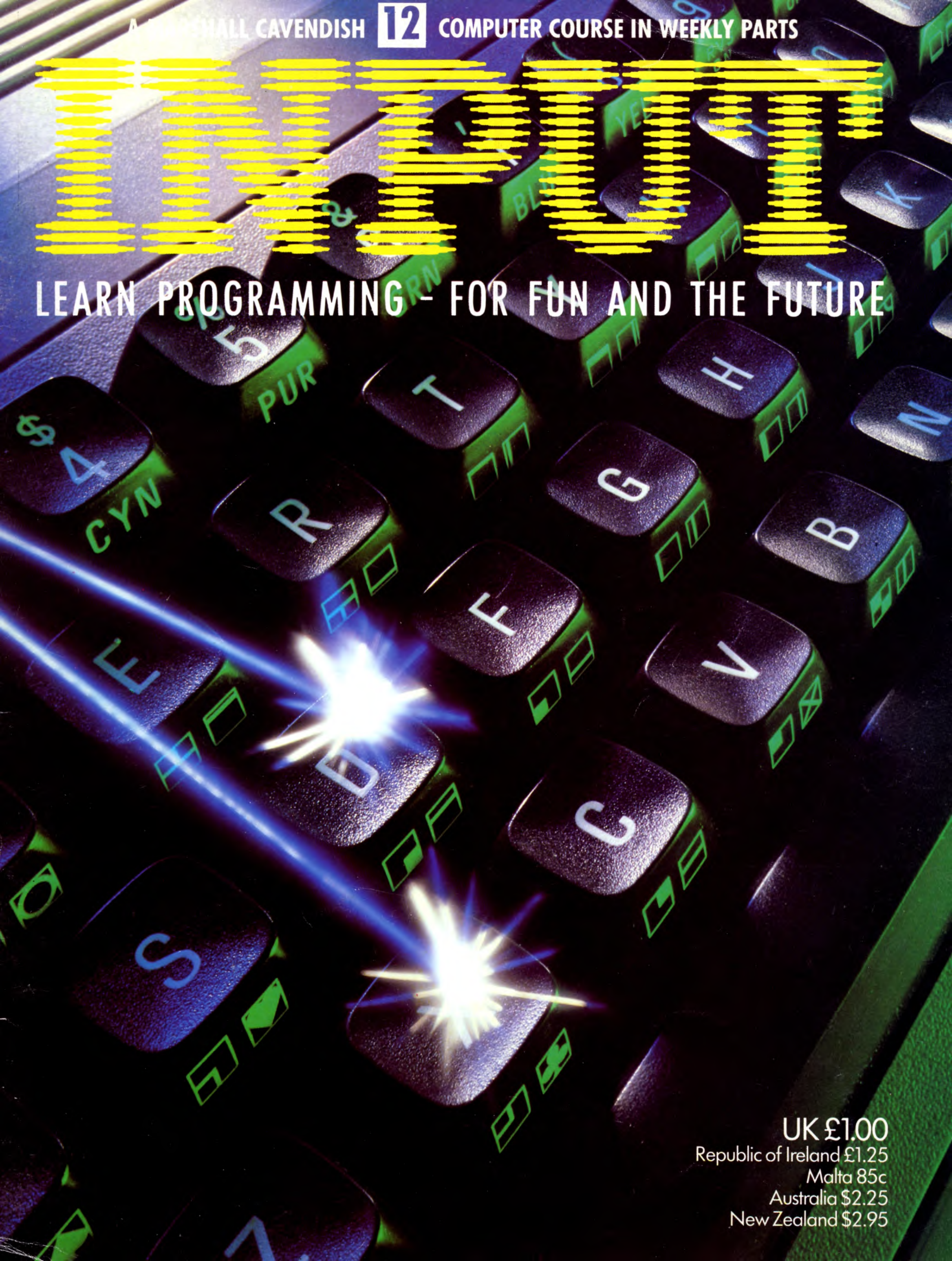
A MARSHALL CAVENDISH **12** COMPUTER COURSE IN WEEKLY PARTS

INPUT

LEARN PROGRAMMING - FOR FUN AND THE FUTURE



UK £1.00
Republic of Ireland £1.25
Malta 85c
Australia $2.25
New Zealand $2.95

INPUT

Vol. 1 No 12

INDEX

The last part of INPUT, Part 52, will contain a complete, cross-referenced index. For easy access to your growing collection, a cumulative index to the contents of each issue is contained on the inside back cover.

PICTURE CREDITS

Front cover, Dave King. Pages 353, 354, Kuo Kang Chen. Page 356, Steve Cross. Page 359, Malcolm Harrison. Pages 360, 362, 365, Neil Winstanley. Page 366, Nigel Snowden/Ian Stephen. Pages 367, 369, 372, Peter Reilly. Page 369, ZEFA/Ian Stephen. Pages 370, 371, 372, Marshall Cavendish/Ian Stephen. Pages 375, 376, 377, 378, 379, Paddy Mounter. Page 380, Paul Chave. Pages 382, 384, Digital Arts.

Published by Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA, England. Printed by Artisan Presss, Leicester and Howard Hunt Litho, London.



There are four binders each holding 13 issues.

HOW TO ORDER YOUR BINDERS

UK and Republic of Ireland:
Send £4.95 (inc p & p) (IR£5.95) for each binder to the address below:
Marshall Cavendish Services Ltd, Department 980, Newtown Road, Hove, Sussex BN3 7DN

Australia: See inserts for details, or write to INPUT, Gordon and Gotch Ltd, PO Box 213, Alexandria, NSW 2015

New Zealand: See inserts for details, or write to INPUT, Gordon and Gotch (NZ) Ltd, PO Box 1595, Wellington

Malta: Binders are available from local newsagents.

BACK NUMBERS

Back numbers are supplied at the regular cover price (subject to availability).

UK and Republic of Ireland:
INPUT, Dept AN, Marshall Cavendish Services, Newtown Road, Hove BN3 7DN

Australia, New Zealand and Malta:
Back numbers are available through your local newsagent.

COPIES BY POST

Our Subscription Department can supply copies to any UK address regularly at £1.00 each. For example the cost of 26 issues is £26.00; for any other quantity simply multiply the number of issues required by £1.00. Send your order, with payment to:

Subscription Department, Marshall Cavendish Services Ltd, Newtown Road, Hove, Sussex BN3 7DN

Please state the title of the publication and the part from which you wish to start.

HOW TO PAY: Readers in UK and Republic of Ireland: All cheques or postal orders for binders, back numbers and copies by post should be made payable to: *Marshall Cavendish Partworks Ltd.*

QUERIES: When writing in, please give the make and model of your computer, as well as the Part No., page and line where the program is rejected or where it does not work. We can only answer specific queries – and please do not telephone. Send your queries to INPUT Queries, Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA.

INPUT IS SPECIALLY DESIGNED FOR:

The SINCLAIR ZX SPECTRUM (16K, 48K, 128 and +), COMMODORE 64 and 128, ACORN ELECTRON, BBC B and B+, and the DRAGON 32 and 64.

In addition, many of the programs and explanations are also suitable for the SINCLAIR ZX81, COMMODORE VIC 20, and TANDY COLOUR COMPUTER in 32K with extended BASIC. Programs and text which are specifically for particular machines are indicated by the following symbols:

SPECTRUM 16K, 48K, 128, and +
COMMODORE 64 and 128
ACORN ELECTRON, BBC B and B+
DRAGON 32 and 64
ZX81
VIC 20
TANDY TRS80 COLOUR COMPUTER

A COMPUTER TYPING TUTOR-3

- ADDING THE NUMBERS
- USING THE SHIFT KEY FOR MORE SYMBOLS
- BUILDING UP A REGULAR RHYTHM
- IMPROVING SPEED AND ACCURACY

This time you can extend your skills to the number keys and all the other symbols on the keyboard. There's also a speed game program to help you improve your typing even more

If you have followed the whole of the typing tutor so far, you will be familiar with all of the letter keys and punctuation on the bottom three rows of the keyboard. But as yet, you won't have touched on the all-important (for program listings) number keys. Nor have you yet learnt how to shift the keys to get either capital or lower case letters—plus the extra punctuation and symbols available.

The programs in this section show you how to add in all of these to your developing typing skills. And in addition there's an exercise which will allow you to increase your speed, accuracy and rhythm—and it's fun too!

ADDING THE NUMBERS

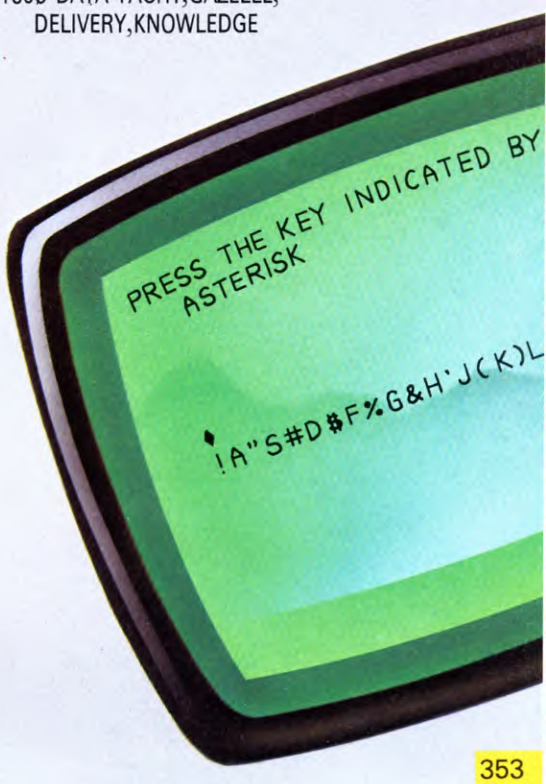
As with the previous two programs (pages 289 to 295 and 328 to 332), adding the number line on the keyboard can be done with simple modifications to the original program you used in the earlier lessons. So LOAD the last program and type in these extra lines. Some of them overwrite existing lines, and some are numbered to slot in amongst those you already have:

```
30 LET S$="1A2S3D4F5G6H7J8K9L0"
210 FOR K=6 TO 24
230 LET R$=S$(K-5)
320 LET RN=INT (RND*19)+1
330 PRINT AT 10,RN+5;"*": LET
    R$=S$(RN)
350 PRINT AT 10,RN+5;"□"
440 LET RN=INT (RND*19)+1
530 PRINT AT 10,13;"□□□□□
    □□□□□": PRINT AT 10,13;T$
540 FOR M=1 TO LEN T$: PRINT AT 9,
    11+M;"□*□□□□□□□□
    □□□"
610 FOR N=1 TO 5: RESTORE : LET
    RN=INT (RND*24)+1:FOR K=1 TO RN:
    READ X$: NEXT K
1010 PRINT AT 12,6;S$
```

```
2000 DATA "MC6809E","VALUE",
    "LAST","Z80A","RELAX",
    "6502","A37XZ","1024",
    "VASTLY","JUNK","RETURN",
    "67VDG"
2010 DATA "ONLY","EXCITED",
    "74LS83","ROUND","LINER",
    "1984","30123","FROWN",
    "4MHZ","WX101","64MB",
    "VIDEO"
```

```
40 A$="1A2S3D4F5G6H7J8K9L0:":
    POKE 54296,15:GOTO 380
80 PRINT "■■■■■■";:
    IF K<3 THEN PRINT"■■"
    TAB(10)A$:POKE 198,0
90 X=INT(RND(1)*20)+1:N=N+1:
    P=P+1:IF K=1 THEN X=N:
    GOTO 120
100 IFK=3 THEN X=INT(RND(1)*20)
    +1:PRINTTAB(18)"■■■■■
    ■■■■■■■"MID$(A$,X,1)
    "■■■■■■■■■■■■"
110 IF K=4 THEN PRINT TAB(16)
    "■□□□□□□□□□□□□
    ■■■■■■■■■■■■■■■■■■■■"W$
    (WW):X=N+6
120 IF K<3 OR K=4 THEN PRINT
    "■■■■■"TAB(10+(X-1))
    "■*"
220 IF K<3 OR K=4 THEN PRINT
    "■■■■■"TAB(10+
    (X-1))"□"
260 IF K=1 AND N<20 THEN 70
540 DATA R2D2,VIC20,COM64,1A2B,
    28.3.67,AGE16,7UP,YOU2,
    A37NO,1SEC
550 DATA 9H53B,16/92,01.P5V,
    19TO572,STEP.73,1984,ZX81,
    13/3/84,WE4ARE1
560 DATA S7T4N,Q3C457,A13579Z,
    P14A41,6K82F3,S0X7,SO52ME,
    D00R.,6502CHIP
```

```
20 DIM A$(40)
30 FOR T=1 TO 40:READ A$(T):
    NEXT
40 A$="1A2S3D4F5G6H7J8K9L0;-:":
    A2$=A$
420 PRINTTAB(9,10)A2$
430 FOR T=1 TO 22
460 PRINTTAB(8+T,9)"*"TAB
    (8+T,10)B2$
530 PRINTTAB(8+T,9)"□"TAB
    (8+T,10)B2$
630 PRINTTAB(9,10)A2$
650 P=RND(22):X=P+8
820 P=RND(22):B2$=MID$(A$,P,1)
950 P=RND(40)
980 PRINTTAB(17,14) B2$"□□
    □□"
1150 B2$=A$(RND(40))
1180 B2$=B2$+"□"+
    A$(RND(40))
1330 DATA 3B3N44,B5N7JM,VB35N45,
    3B43BV6NHJ454,FRN312V,1CWNY,
    C43N4HJ45B,4343HG3BN4334,
    "34:;DWDP","D34C,3RUY,O",
    "QAY,9UN,D7B8"
1350 DATA FOUND,QUITE,GREY,
    AZURE,QUICKLY,PRIZE,ZONE,
    COLOUR,ALTHOUGH,CERTIFY
1360 DATA YACHT,GAZELLE,
    DELIVERY,KNOWLEDGE
```



```
10 OB$="1A2S3D4F5G6H7J
   8K9L0;:—"
210 AP=1252
220 FOR K=1 TO 22
320 AP=1252+RND(22)
430 P$=MID$(OB$,RND(22),1)
1020 PRINT@261,OB$
9000 DATA MC6809E,VALUE,"LAST:",
   Z80A,RELAX,6502,A37XZ,—1024,
   VASTLY,JUNK
9010 DATA RETURN,A847VDG,145.22,
   "ONLY,",EXCITED,74LS83,—93.41,
   ROUND,LINER
9020 DATA PROCESS,FROZEN,BRING,
   1984,DRAWN,BLOOD,842.52,"301,
   123",350KG
```

When you RUN the program, you will be asked to select one of the familiar five levels. These are much as before, with the addition of the extra characters. In the lower levels, you will be asked to mix and match numbers, and sometimes punctuation, with the existing keys—this makes it hard for you, by ensuring that you can't just concentrate on the numbers.

Then, in the higher levels, you will be asked to type a mixture of words, groups of numbers, and compounds that mix letters and numbers together. You will see the words and numbers that are selected in the DATA statements near the end of the program. If you wish, you can change these after a time, should you get too familiar with what the computer is going to ask you to do, and want to give yourself a real challenge! But remember to keep the number of words (or groups of characters) the same or the computer will not be trying to READ the correct amount of DATA.

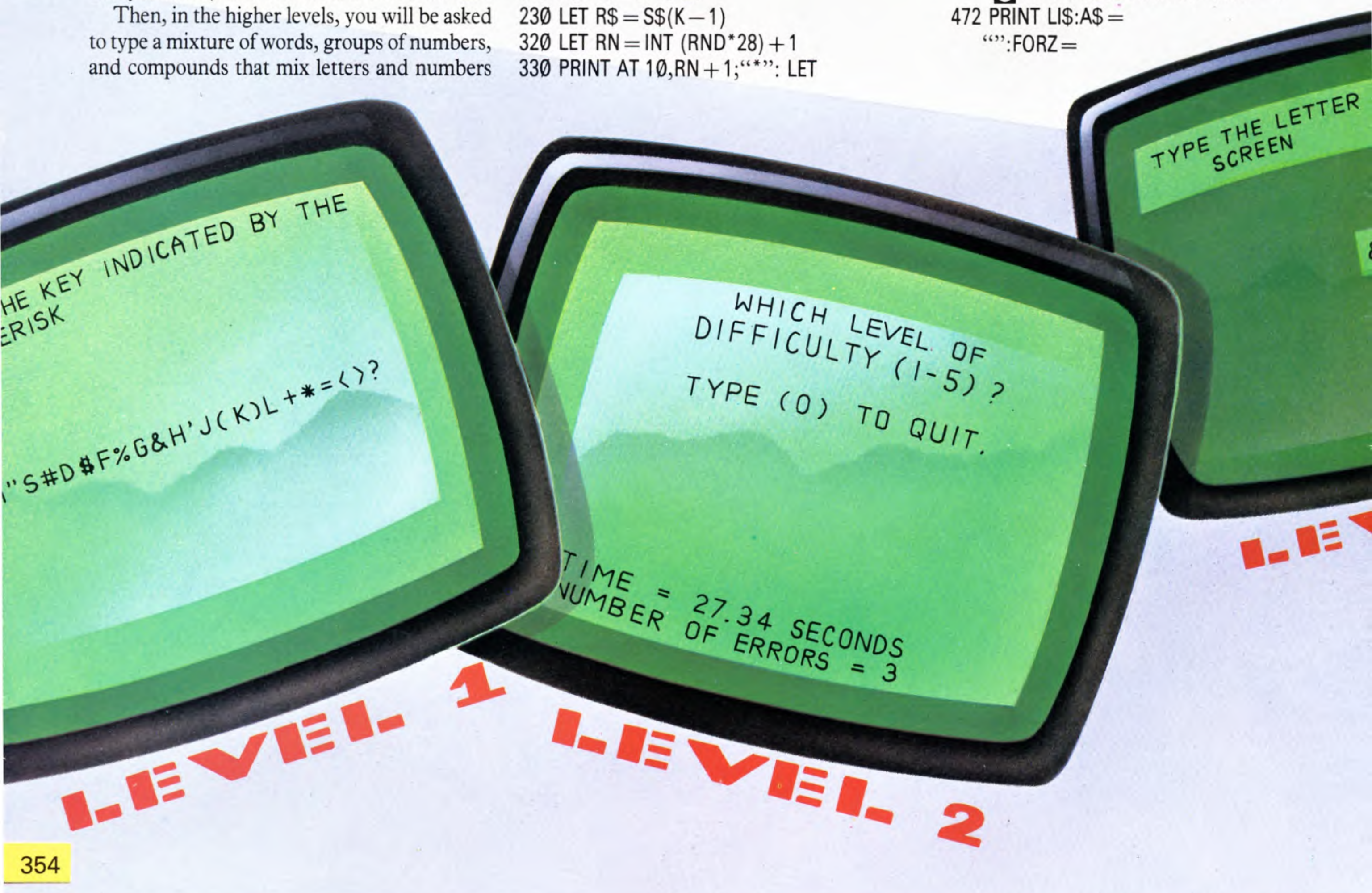
When you have the number keys at your command, it's time to move on to the next lesson. This will give you practice at getting the characters for which you need to press the SHIFT key.

GETTING SHIFTED

This time, the extra program lines will add the ability to use the SHIFT key. As before, they either overwrite the existing lines or add into them:

```
20 POKE 23658,0: LET ER=0
30 LET S$="A!aS@sD#dF$fG%gH
   &hJ'jK(kL)l0"
210 FOR K=2 TO 29
230 LET R$=S$(K—1)
320 LET RN=INT (RND*28)+1
330 PRINT AT 10,RN+1;"*": LET
   R$=S$(RN)
350 PRINT AT 10,RN+1;"□"
440 LET RN=INT (RND*28)+1
610 FOR N=1 TO 4: RESTORE : LET
   RN=INT (RND*24)+1: FOR K=1 TO RN:
   READ X$: NEXT K
1010 PRINT AT 12,2;S$
2000 DATA "$235.50","PRINT#",
   "&H1200","23.5%","Account",
   "LONDON","They're", "15 @ £12",
   "(under)","H+9=1D","**List**",
   "Fire!!"
2010 DATA ";—;;—;","Extra",
   "Out-of","Shall","4*4=16",
   "Why?","6:10pm","We'll","£15.40",
   "Drive", "ATTENTION","100/4"
```

```
40 PRINTCHR$(8)CHR$(14):POKE
   54296,15:GOTO 380
130 IF K=5 THEN FOR Z=1 TO MM:
   PRINT"[π]"W$(Z);:NEXTZ:
   PRINT:PRINT"[▘][◣]";
360 IF K=5 AND WW<>4 THEN W$
   (WW)=W$(WW)+"□"
470 IF K=5 THEN PRINT TAB(12)
   "[■]TYPE THESE WORDS":MM=4
472 PRINT LI$:A$=
   "":FORZ=
```



```
   1TO10:Q1=65+RND(1)*26
473 Q2=35+INT(RND(1)*23)
474 IF RND(1)>.50 THEN Q1=Q1+128
480 A$=A$+CHR$(Q2)+CHR$(Q1):
   NEXT
540 data"$174.374","Don't",
   "And","Account","Kill",
   "87.54%","10,29","hello!"
550 data"(2+6*1-8)","These",
   "17-3-67","Can't","Then",
   "LDA#248","You&Me","Fred"
560 data"**Ravi**","Year'1984",
   "&x)!KQ?","Com-64","#(XxX)#"
570 data"Typing","Tutor","873Pence",
   "(AdGjL)","STA$C0B2",
   "Computer","A$+STR$(1)"
```

```
20 DIM A$(30)
30 FOR T=1 TO 30:READ A$(T):NEXT
40 A$="!A""S#D$F%G&H'J(K)L=;~:
   ¦{£+*<>?":A2$=A$
420 PRINTTAB(6,10)A2$
430 FOR T=1 TO 30
460 PRINTTAB(5+T,9)"*"TAB
   (5+T,10)B2$
530 PRINTTAB(5+T,9)"□"TAB
   (5+T,10)B2$
630 PRINTTAB(6,10)A2$
650 P=RND(30):X=P+5
820 P=RND(94)+32:B2$=CHR$(P)
950 P=RND(30)
980 PRINTTAB(17,14) B2$
   "□□□□□□□□□□□□□□"
1150 B2$=A$(RND(30))
1180 B2$=B2$+"□"+A$(RND(30))
1330 DATA Libraries,Sounds,Input,
   Notebook,Commuting,Will(192),
   A$(102),Stone(13),pounds(89),
   *!%$#*~,"£103,964",Silver,
   Ordering,Price,Socks,Mr.F.Callen,
   Projects,Marker,Hello
1340 DATA %$hh&fljKLUs,'&yttehJK
   YHDS,EP@#9272EJWe,HEGENi
   @@*a{},kjgil~|{£e""YGW
1350 DATA Found,Gazelle,Delivery,
   Words,hfhhFJD,Kdldee#$
```

```
10 OB$="!A"+CHR$(34)+
   "S#D$F%G&H'J(K)L+*=<>?"
20 POKE329,0:CLS
210 AP=1250
220 FOR K=1 TO 24
320 AP=1250+RND(24)
430 P$=MID$(OB$,RND(24),1)
999 POKE329,255:CLS:END
1020 PRINT@259,OB$
9000 DATA PRINT#,Shown,&H4000,
   Out!!,(under),H+9=1D,$500.
   10,D/100 %,They're,**list**
9010 DATA Extra,Charge,DAILY,
   Account, Month,Reply,Today,
   Manage,Section
9020 DATA LONDON,Shall,Touch,;
   -;;-;,Success,Out-of,
   Replace,Country,Drive
```





Note, on the Tandy you'll have to change the 329 to 282 in Lines 20 and 999.

On the Dragon and Tandy, you cannot PRINT lower case characters on the screen. Instead these are displayed as an inverse character—■ instead of S, for example. It may take some time to get used to these.

On the Commodore change to lower case/graphics mode to enter the DATA statements.

The lower levels of the test now present you with all the characters which are only available when the keys are shifted—punctuation, mathematical symbols etc. To make it harder for you, they are mixed in with the letters of the home keys, which means you have to return your fingers every time.

On the higher levels, you now get a list of words and groups of characters as before, except that this time you will find capital letters and shifted symbols mixed in with the lower case. Your computer will test your speed and accuracy on these words and character groups. And if you find that you get too good at these particular test examples, you can always give yourself a new set of DATA. You must remember to keep the total number of DATA entries the same, however. When you can find all of the keyboard characters—without looking and without hesitating—you can move on to the next section.



SPEED GAME

Now's the time to think about improving your accuracy and speed. One of the best ways to do this is to type to the beat of a metronome, or something similar, which will improve your regularity and rhythm. Then, as you get more proficient, you can speed up the metronome—and hence your typing.

But why bother using a metronome, when your computer has a built-in clock? The next program is a complete, new typing exercise that is laid out like a game—where your score depends on how good you are with the keyboard. It is in two parts. The first part displays a line of characters selected at random—you have to type the characters as they appear in sequence. The second part is harder—because now the characters are thrown up on the screen at random, one by one, so you have no clue as to what is next.

Before the test starts, you can select your own level of difficulty. This is done by telling

the computer how fast you want the letters to be displayed—in other words, how many characters in a minute you want to type. The computer will then set you a limited time within which you have to type each character, or else you will be given an error score. On the first level this is done by a moving indicator which shows you which letter you should be typing, and on the second level it is done by flashing up the character for a set time only.

Before you start the first test, you can choose whether you want the normal keyboard (letters only) or extended keyboard (all symbols, too). Also before you start the second level (the characters test), you can decide how long you can keep it up. You will be asked how many characters in total you want to be in the test. When the computer has displayed them all, it will stop and give you a score based on your errors.

It's not just overall speed that counts to beat the computer at this challenging test, because you need to build up a steady rhythm. This will be of real benefit to anyone who wishes to increase their general typing speed. To help you to gain the habit, the computer will give you a sound signal as well as the visual prompt for each letter.

Now type in the program itself, and put your skills to the test:

```
10 BORDER 7: PAPER 7: INK 0: CLS
20 LET a$ = "ABCDEFGHIJKLMNOPQRS
   TUVWXYZ"
30 LET a$ = a$ + "abcdefghijklmnop
   qrstuvwxyz"
40 LET a$ = a$ + "1234567890!@#$%&'()"
50 LET a$ = a$ + CHR$ 34 + "<>;-+
   =£?/*,."
60 PRINT INVERSE 1;AT 6,7;"□TEST 1 OR
   TEST 2□"
70 IF INKEY$ = "" THEN GOTO 70
80 LET i$ = INKEY$: IF i$ = "2" THEN GOTO
   400
90 IF i$ <> "1" THEN GOTO 70
100 CLS : INPUT "How many characters a
    minute?□";cpm
110 LET t = 3000/cpm
120 LET s$ = ""
130 FOR n = 1 TO 30
140 LET s$ = s$ + a$(INT (RND*84) + 1)
150 NEXT n
160 PRINT BRIGHT 1;AT 11,1;s$
200 GOSUB 800: LET er = 0: FOR r = 1 TO 30
210 POKE 23672,0: POKE 23673,0
220 PRINT AT 10,r-1;"□*"
230 BEEP .02,20
240 IF PEEK 23672 + 256*PEEK 23673 >= t
    THEN LET er = er + 1: GOTO 300
250 LET i$ = INKEY$: IF i$ = "" THEN GOTO
    240
260 IF i$ = s$(r) THEN PRINT AT 12,r;"↑":
    GOTO 280
270 LET er = er + 1
280 IF PEEK 23672 + 256*PEEK 23673 < t
    THEN GOTO 280
300 NEXT r
```



```
310 PRINT AT 16,3;"YOU GOT□";er;"□OUT
    OF 30 WRONG"
320 FOR f=1 TO 200: NEXT f
330 GOTO 20
400 CLS : INPUT "Number of keys a
    minute?□";cpm
410 INPUT "Number of characters?□";r
420 INPUT "(N)ormal or (E)xtended keys?□";
    LINE m$
430 IF m$="N" OR m$="n" THEN LET
    a$=a$( TO 52): GOTO 450
440 IF m$<>"E" AND m$<>"e" THEN
    GOTO 420
450 LET t=3000/cpm
460 GOSUB 800
470 LET er=0
480 FOR n=1 TO r
490 POKE 23672,0: POKE 23673,0
500 LET r$=a$(INT (RND*LEN a$)+1)
510 PRINT INVERSE 1;AT 10,15;r$; INVERSE
    0;AT 11,15;"□"
520 BEEP .02,20
530 IF PEEK 23672+256*PEEK 23673>=t
    THEN LET er=er+1: GOTO 580
540 LET i$=INKEY$: IF i$="" THEN GOTO
    530
550 IF i$=r$ THEN PRINT AT 11,15;"↑":
    GOTO 570
560 LET er=er+1
570 IF PEEK 23672+256*PEEK 23673<t
    THEN GOTO 570
580 NEXT n
590 PRINT AT 16,3;"YOU GOT□";er;"□OUT
    OF□";n-1;"□WRONG"
600 FOR f=1 TO 200: NEXT f
610 GOTO 20
800 LET c$="5..4..3..2..1..0"
810 FOR n=1 TO 16
820 PRINT AT 2,5+n;c$(n)
830 PAUSE 10
840 NEXT n
850 BEEP .2,10
860 PRINT AT 2,0;TAB 31;"□"
870 RETURN
```



```
10 FOR Z=1TO40:LI$=LI$+"—":
   NEXT:LI$="■"+LI$
20 PRINT "■■"TAB(7)
   "******TYP■ING ■TUT■OR*****"
30 PRINTCHR$(8)CHR$(14):
   POKE 54296,15:GOTO 260
40 S=0:WW=1:ER=0:NM=0:N=0
50 POKE53280,5
60 PRINT "■■■■■■";:
   IF K=1 THEN PRINT"■■"
   TAB(10)A$:POKE198,0
70 X=INT(RND(1)*20)+1:N=N+
   1:P=P+1:IF K=1 THEN X=N:GOTO 90
80 IFK=2THENX=INT(RND(1)*20)
   +1:PRINTTAB(18)"■■■■
   ■■■■■■■■"MID$(A$,
   X,1"■■■■■■■■■■■■"
90 IF K=1 THEN PRINT "■■■
   ■■"TAB(10+(X-1))"■*"
100 TI$="000000"
110 GET K$:IF K$="" AND TI<TM THEN
    110
120 IF TI>TM THEN ER=ER+1:
    GOTO170
130 IFK$<>MID$(A$,X,1) THEN 110
140 IFK=1THENFORZ=4TO1STEP-1:
    PRINTLEFT$("■■■■■
    ■■■■",Z+6)TAB(10+
    X-1)"■ ✝ ■■□":NEXT
150 IFK=2THENPRINT"■■■■
    ■■"TAB(18)"■■■■
    ■■■■■■■■■■■■■
    ■■■■■■■■■"
160 IF TI<TM THEN 160
170 W=54276:A=54277:NM=NM+1:
    POKE 53280,6
180 IF K=1 THEN PRINT"■■■
    ■■"TAB(10+(X-1))"□"
190 POKE W,33:POKE A,50:
    POKE 54273,150
200 POKE 54273,0:POKE W,32
210 IF K=1 AND N<20 THEN 50
220 IF K=2 AND N<TL THEN 50
230 PRINT "■■"LI$"■AT"KP
    "KEY PRESSES PER MINUTE YOU GOT"
240 PRINT ER"OUT OF"NM"WRONG"
250 PRINT LI$
260 POKE 53280,6:POKE 53281,0:
    POKE 198,0
270 PRINT "■■■■■■■■■
    ■■■"TAB(15)"OPTIONS■■"
280 FOR Z=1 TO 2:PRINT TAB
    (13)Z;":TEST";Z:NEXT Z
290 PRINT TAB(12)"■■ENTER CHOICE?"
300 GET K$:K=VAL(K$):IF K<1 OR K>2
    THEN 300
310 INPUT"■INPUT KEY PRESS PER
    MINUTE";KP:IFKP<1THEN310
320 TM=3000/KP:INPUT"■NORMAL OR
    EXTENDED KEYS (N/E)";NX$
330 IF NX$<>"N" AND NX$<>"E"
    THEN 320
340 IF K=2 THEN TL=20:INPUT
    "■INPUT NUMBER OF CHARACTERS";TL
350 PRINT"■"LI$"■"
360 PRINT "■ TYPE THE LETTER WHEN YOU
    HEAR THE BEEP"
370 SP=1:IF NX$="N"THEN SP=.5
380 PRINT LI$:A$="":FORZ=1
    TO10.5STEP SP:Q1=65+
    RND(1)*26
390 Q2=33+INT(RND(1)*25)
400 IF RND(1)>.50 THEN Q1=
    Q1+128
410 IF NX$="N"THENA$=A$+
    CHR$(Q1):NEXT:GOTO430
```

```
420 A$=A$+CHR$(Q2)+CHR$(Q1):
    NEXT
430 PRINT TAB(11)"■PRESS KEY TO
    START":POKE 198,Ø:WAIT 198,1
440 PRINT TAB(11)"◘□□□□
    □□□□□□□□□□□□
    □□□"
450 GOTO 4Ø
```

```
10 MODE1
20 *FX2Ø2,48
30 VDU23;82Ø2;Ø;Ø;Ø;
40 CLS:INPUT'''"WHICH TEST 1 OR 2",G
50 IF G<>1 AND G<>2 THEN 4Ø
60 CLS:INPUT'''"(N)ORMAL OR
   (E)XTENDED",C$
70 IF C$="n" THEN C$="N"
80 ON G GOTO 9Ø,12Ø
90 PROCFIR
100 *FX15,1
110 GOTO 4Ø
120 PROCSEC
130 *FX15,1
140 GOTO 4Ø
150 DEF PROCFIR
160 B=Ø:F=1
170 CLS:PRINT'"LINE TEST":A$=""
180 INPUT''"RATE OF KEY PRESSES PER
    MINUTE",R:TF=6ØØØ/R
190 FOR T=1 TO 3Ø
200 IF C$="N" THEN A$=A$+
    CHR$(RND(26)+INT(RND(1)+.5)
    *32+64):GOTO 22Ø
210 A$=A$+CHR$(RND(94)+32)
220 NEXT
230 PROCGO
240 PRINTTAB(5,15)A$
250 N=3Ø
260 FOR T=1 TO 3Ø:X=T+4
270 PRINTTAB(X,14);"*"
280 B2$=MID$(A$,T,1)
290 PROCTIME
300 PRINTTAB(X,14);"□"
310 NEXT
320 PROCMES
330 ENDPROC
340 DEF PROCSEC
350 B=Ø:F=Ø
360 CLS:PRINT'"SINGLE CHARACTER TEST"
370 INPUT''"RATE OF KEY PRESSES PER
    MINUTE",R:TF=6ØØØ/R
380 INPUT''"NUMBER OF CHARACTERS
    WANTED",N
390 PROCGO
400 X=19
410 FOR T=1 TO N
420 IF C$="N" THEN B2$=CHR$
    (RND(26)+INT(RND(1)+.5)*32
    +64):GOTO 44Ø
430 B2$=CHR$(RND(94)+32)
440 PRINTTAB(19,15)B2$
450 PROCTIME
460 NEXT
470 PROCMES
480 ENDPROC
490 DEF PROCMES
500 FOR T=1 TO 2ØØØ:NEXT
510 CLS:PRINTTAB(Ø,1Ø)"YOU
    GOT□";B;"□OUT OF□";N;"□RIGHT"
520 PRINT''"YOUR TYPING RATE
    WAS□";R;"□PER MIN"
530 INPUT"PRESS RETURN FOR NEXT TEST"A
540 ENDPROC
550 DEF PROCTIME
560 IF F=Ø THEN PRINTTAB(X,16)"□"
570 *FX15,1
580 TIME=Ø
590 SOUND1,-15,15Ø,1
600 B$=INKEY$(1)
610 IF B$=B2$ THEN B=B+1:
    PRINTTAB(X,16)"^"
620 IF TIME>TF THEN 64Ø
630 IF B$="" THEN 6ØØ
640 PRINTTAB(X,15)"□"
650 IF TIME<TF THEN 65Ø
660 ENDPROC
670 DEF PROCGO
680 CLS:PRINT''"PRESS THE KEY WHEN YOU
    HEAR THE BEEP"
690 PRINTTAB(Ø,1Ø);
700 FOR T=5 TO 1 STEP-1
710 A=INKEY(1Ø):PRINT;T;
720 FOR P=1 TO 3:A=INKEY(1Ø):
    PRINT".";:NEXT
730 NEXT
740 A=INKEY(25):PRINT"GO"
750 A=INKEY(1):A=INKEY(5Ø)
760 PRINTTAB(Ø,1Ø)"□□□□□□□
    □□□□□□□□□□□□□□□□□"
770 ENDPROC
```

Microtip

Speeding up

When you're using the speed game program it's best to start with the normal keyboard and a slow speed, say between 30 and 50 characters per minute.

The danger to avoid is typing the familiar characters quickly and then slowing down while you look for the characters you're less sure of—particularly if you choose the extended keyboard.

Once you get the hang of typing to a constant rhythm you can select the extended keyboard and then gradually increase your speed.

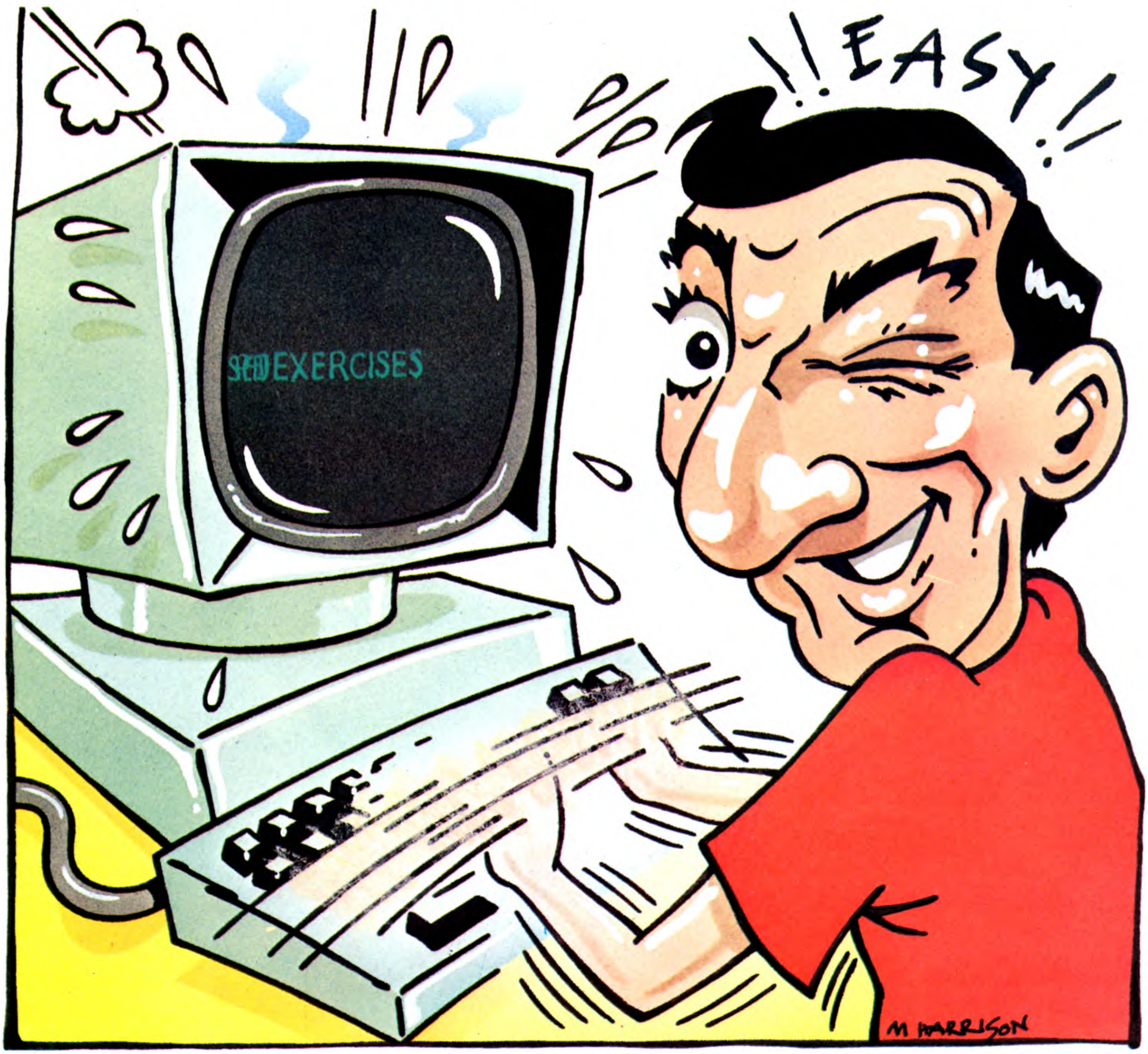
```
10 CLS
20 PRINT@7Ø,"WHICH TEST (1 OR 2) ?"
30 PRINT@1Ø4,"TYPE (Ø) TO QUIT"
40 A$=INKEY$:IFA$<"Ø" OR A$>"2"
   THEN 4Ø
50 ON VAL (A$)+1 GOSUB 1ØØØ,6ØØ,2ØØ
60 POKE 329,255
70 ER=Ø:W$="":B$=""
80 GOTO 2Ø
200 CLS:INPUT"INPUT KEY PRESSES PER
    MINUTE";KP
210 IF KP<1 THEN 2ØØ
220 INPUT"INPUT NUMBER OF
    CHARACTERS□";NC
230 IF NC<1 THEN 22Ø
240 NM=NC
250 PRINT:PRINT"NORMAL OR EXTENDED
    KEYS (N/E) ?"
260 A$=INKEY$:IF A$<>"N" AND
    A$<>"E" THEN 26Ø
270 RN=9Ø:ST=32:IF A$="N" THEN
    RN=58:ST=64
280 POKE 329,Ø
290 TM=3ØØØ/KP
300 CLSØ:PRINT"□PRESS THE KEY AFTER
    THE BEEP□"
310 PRINT @238,"□□□";:PRINT@27Ø,
    "□□□";:PRINT@3Ø2,"□□□";
320 W$=CHR$(RND(RN)+ST)
330 IF W$>"Z" AND W$<"a" THEN 32Ø
340 PRINT@271,W$;
350 TIMER=Ø
360 A$=INKEY$:IF A$="" THEN 38Ø
370 B$=A$:IF B$=W$ THEN SCREENØ,1
380 IF TIMER<TM THEN 36Ø
390 SOUND15Ø,1:IF B$="" THEN 35Ø
400 W$=CHR$(RND(RN)+ST)
410 IF W$>"Z" AND W$<"a" THEN 4ØØ
420 PRINT@271,W$;
430 TIMER=Ø
440 A$=INKEY$:IF A$="" THEN 46Ø
450 B$=A$:IF B$=W$ THEN
    SCREENØ,1
460 IF TIMER<TM THEN 44Ø
470 SOUND15Ø,1
480 IF B$<>W$ THEN
    ER=ER+1:GOTO49Ø
490 POKE1295,128
500 NC=NC-1:IF NC>Ø THEN 4ØØ
510 CLS:PRINT@448,"AT";KP;
    "KEY PRESSES PER MINUTE"
520 PRINT"YOU GOT";ER;"OUT
    OF";NM;"WRONG";
530 RETURN
600 CLS:INPUT"INPUT KEY PRESSES PER
    MINUTE";KP
610 IF KP<1 THEN 6ØØ
620 PRINT:PRINT"NORMAL OR EXTENDED
    KEYS (N/E) ?"
```

```
630 A$=INKEY$:IF A$<>"N" AND
    A$<>"E" THEN 63Ø
64Ø RN=91:ST=31:IF A$="N" THEN
    RN=58:ST=64
65Ø TM=3ØØØ/KP
66Ø CLS:POKE 329,Ø
67Ø FOR K=1 TO 32
68Ø CR=RND(RN)+ST
69Ø IF CR>9Ø AND CR<97 THEN CR=32
7ØØ W$=W$+CHR$(CR)
71Ø NEXT
72Ø AP=1248
73Ø POKE AP,1Ø6
74Ø PRINT@256,W$
75Ø TIMER=Ø
76Ø A$=INKEY$:IF A$="" THEN 78Ø
77Ø B$=A$
78Ø IF TIMER<TM THEN 76Ø
79Ø SOUND 15Ø,1:IFB$="" THEN 75Ø
8ØØ TIMER=Ø
81Ø A$=INKEY$:IF A$="" THEN 83Ø
82Ø B$=A$
83Ø IFTIMER<TM THEN 81Ø
84Ø SOUND 15Ø,1
85Ø IF B$<>MID$(W$,AP-1247,1) THEN
    ER=ER+1:GOTO 86Ø
855 POKE AP+64,94
86Ø POKE AP,96
87Ø AP=AP+1
88Ø IF AP=128Ø THEN 91Ø
89Ø POKE AP,1Ø6
9ØØ B$="":GOTO 8ØØ
91Ø CLS:PRINT@481,"AT";KP;
    "KEYS PER MINUTE"
92Ø PRINT" YOU GOT";ER;"OUT OF 32
    WRONG !";: RETURN
1ØØØ CLS
```

Again, on the Tandy change the 329 to 282 in Lines 6Ø, 28Ø and 66Ø.

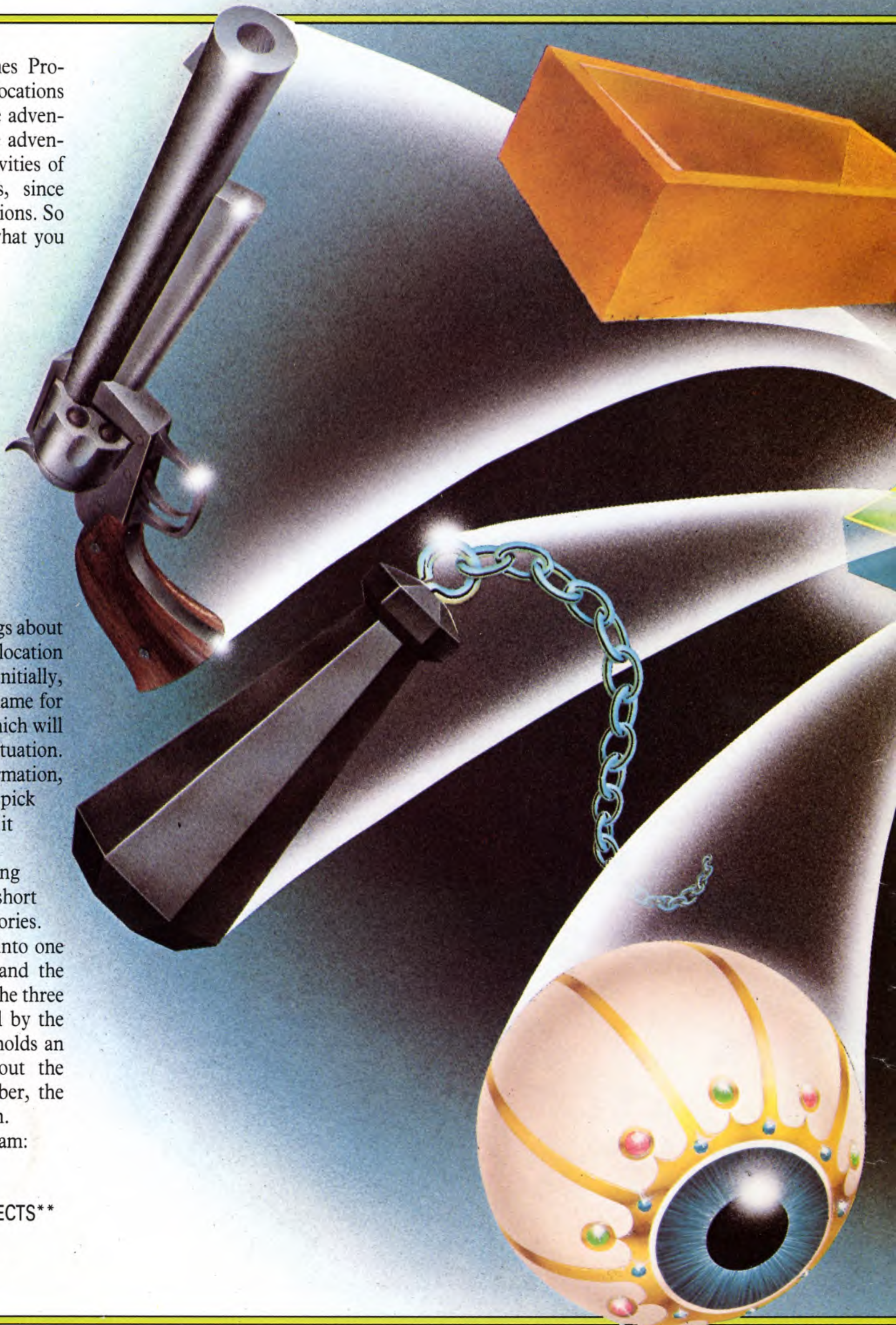
TAKING THINGS FURTHER

Practise all the exercises so far, and you will have a thorough knowledge of all the character keys. This part of the course is self-contained in itself. However, in a later article, you will get an opportunity to practise your skills on some real sentences, and see how to lay out a neat passage of typing.



THE OBJECTS OF THE QUEST

Now is the time to fill your empty adventure world with objects. We show you how to enter your list of items into the program, and then how to manipulate them



At the end of the last part of Games Programming you had a complete set of locations for your adventure and had given the adventurer the ability to wander round the adventure world. But at this stage, the activities of the adventurer are rather pointless, since nothing yet happens in any of the locations. So now is the time to go back and see what you had planned to include at each point.

Now you'll see how to add some more routines which will put all the objects which make up the quest in the right place. Other routines will allow the adventurer to collect the objects or leave them behind. You'll also type in a routine which will list an inventory of all the objects being carried—very useful when faced with a problem.

LOAD the program from last time, ready to receive the new routines.

OBJECTS

The machine needs to know three things about the objects in the adventure: the location number where the object is placed initially, ready for the adventurer to find it, a name for the object, and a longer description which will include something about the object's situation. You need all three pieces of information, because firstly the computer needs to pick an object to suit each location. Then it will need to tell the adventurer what is in the location—hence using the long description. And finally, it needs the short name for use in instructions or inventories.

The location numbers will be fed into one array, the object title into another, and the longer descriptions into yet another. The three arrays will be manipulated in parallel by the program—each element of the array holds an equivalent piece of information about the objects, the first is the location number, the second is the object's name, and so on.

Add these extra lines to your program:

S

```
160 REM **SET UP ARRAYS FOR OBJECTS**
170 READ NB
180 DIM B(NB): DIM B$(NB,14): DIM
    S$(NB,40)
```

- ENTERING THE DATA STATEMENTS FOR THE OBJECTS
- SHORT AND LONG DESCRIPTIONS
- PRINTING THE OBJECTS IN THE CORRECT LOCATIONS
- MORE VERBS
- CHOOSING THE RIGHT ROUTINE
- PICKING UP AND DROPPING THE OBJECTS
- ADDING AN INVENTORY ROUTINE



```
190 FOR I=1 TO NB: READ B(I),B$(I),S$(I):
    NEXT I
200 DATA 7,4,"BAG","A BAG OF MARBLES
    IS HERE"
210 DATA 14,"BRICK","A BRICK LIES ON
    THE GROUND"
220 DATA 24,"CHAIN","THERE IS A CHAIN
    HANGING"
230 DATA Ø,"GUN","THERE IS A GUN ON
    THE FLOOR"
240 DATA Ø,"EYEBALL","A JEWELLED
    EYEBALL LIES ON THE□□GROUND"
250 DATA 22,"LAMP","YOU SEE A LAMP"
260 DATA Ø,"TAX INSPECTOR","A TAX
    INSPECTOR SUDDENLY APPEARS"
```

```
160 REM **SET UP ARRAYS FOR OBJECTS**
170 READ NB
180 DIM OB(NB),OB$(NB),SI$(NB)
190 FOR I=1 TO NB: READ
    OB(I),OB$(I),SI$(I):NEXT
200 DATA 7,4,BAG, A BAG OF MARBLES IS
    HERE
210 DATA 14,BRICK,A BRICK LIES ON THE
    GROUND
220 DATA 24, CHAIN,THERE IS A CHAIN
    HANGING
230 DATA Ø,GUN,THERE IS A GUN ON THE
    FLOOR
240 DATA Ø,EYEBALL,A JEWELLED EYEBALL
    LIES ON THE GROUND
250 DATA 22,LAMP,YOU SEE A LAMP
260 DATA Ø,TAX INSPECTOR,A TAX
    INSPECTOR SUDDENLY APPEARS
```

```
160 REM **SET UP ARRAYS FOR OBJECTS**
170 READ NB
180 DIM OB(NB),OB$(NB),SI$(NB)
190 FOR I=1 TO NB:READ
    OB(I),OB$(I),SI$(I):NEXT
200 DATA 7,4,BAG, A BAG OF MARBLES IS
    HERE
210 DATA 14,BRICK,A BRICK LIES ON THE
    GROUND
220 DATA 24,CHAIN,THERE IS A CHAIN
    HANGING
230 DATA Ø,GUN,THERE IS A GUN ON THE
    FLOOR
240 DATA Ø,EYEBALL,A JEWELLED EYEBALL
```



```
    LIES ON THE□ □GROUND
25Ø DATA 22,LAMP,YOU SEE A LAMP
26Ø DATA Ø,TAX INSPECTOR,A TAX
    INSPECTOR SUDDENLY APPEARS
```

Each of the Lines from Line 2ØØ to Line 26Ø contains the three pieces of DATA referring to the same object. Line 2ØØ has one extra piece of DATA in the list. The figure 7—the first piece of DATA—tells the machine how many sets of DATA there are.

Once the number 7 has been READ by Line 17Ø, three arrays are DIMensioned to that size by Line 18Ø.OB will contain the location of each object—either a location number, or Ø if the object doesn't yet exist (in other words, things like the eyeball, which have to be uncovered during the adventure), or −1 if it is being carried by the adventurer. OB$ will contain the short descriptions, and SI$ will contain the longer ones.

Line 19Ø fills the arrays with DATA from Lines 2ØØ to 26Ø. The DATA is arranged in sets of three as follows: location number, short description of the object, long description of the object.

When you use this routine in other adventures, you will not have to make too many alterations to its structure because adjusting the first piece of DATA will automatically take care of the size of the FOR . . . NEXT loops and the array DIMensions.

PLANTING THE OBJECTS

The program now contains all the information about what all the objects are and where they are to be put. The next routine displays the longer object description at the appropriate location:

```
36Ø REM **TO PRINT OBJECT IN
    APPROPRIATE LOCATION**
37Ø FOR I=1 TO NB: IF B(I)=L THEN PRINT
    S$(I)
38Ø NEXT I
```

```
36Ø REM **TO PRINT OBJECT IN
    APPROPRIATE LOCATION**
37Ø FOR I=1 TO NB:IF OB(I)=L □THEN
    PRINT SI$(I)
38Ø NEXT
```

At this stage make a small alteration to Lines 33Ø and 34Ø: change GOTO 4ØØ to GOTO 37Ø. Lines 37Ø and 38Ø check through the object location array. If any of the location numbers match the current location—L—then the short description is displayed following the location description. This routine can be used without alteration in other adventures.

MORE VERBS

The adventure now has some objects scattered around its locations, but because the machine doesn't yet understand any words but NORTH, SOUTH, EAST and WEST, the poor adventurer can't do anything with them. Imagine the frustration of not being able to collect that very desirable bag of marbles or to defend yourself against the tax inspector! So you need to give the computer a vocabulary of words it can recognize, telling it what to do with the objects. Later, you'll see what to do if the player enters a word that isn't in the vocabulary you have programmed into the machine.

As the program treats all the direction words as verbs, the best place for the verbs describing what to do with the objects will be in the array R$, and the best place for the corresponding numbers will be in R.

You'll therefore have to make a few alterations starting with Line 13Ø. The limits of the FOR ... NEXT loop will have to be changed. Either type in the whole line afresh or use the machine's editor to change the existing line. Whichever way you choose to do it, Line 13Ø should now read:

```
13Ø FOR K=1 TO 19: READ R$(K),
   R(K): NEXT K
```

```
13Ø FOR K=1 TO 19:READ R$(K),
   R(K):NEXT
```

Now add Lines 14Ø and 145:

```
14Ø DATA "SWIM",5,"EMPTY",6,
   "LIGHT",7,"QUIT",8,
   "INVENTORY",9,"KILL",1Ø,
   "SHOOT",1Ø,"HELP",11
145 DATA "GET",2,"TAKE",2,
   "CARRY",2,"PUT",3,"LEAVE",3,
   "DROP",3,"PULL",4
```

```
14Ø DATA SWIM,5,EMPTY,6,LIGHT,7,
   QUIT,8,INVENTORY,9,KILL,1Ø,
   SHOOT,1Ø,HELP,11
145 DATA GET,2,TAKE,2,CARRY,2,
   PUT,3,LEAVE,3,DROP,3,PULL,4
```

Each verb has a corresponding number. Verbs with the same number have the same meaning as far as the computer is concerned, and will perform the same operation. Designing the program so that it will understand GET, TAKE and CARRY, for example, will save the adventurer wasting time needlessly trying to discover which of these words to use. You can easily add your own words to the DATA lines by changing the FOR ... NEXT loop in Line 13Ø and tacking the extra DATA on the end of Line 145. You'll have to make a number of alterations elsewhere in the program, but in a later part of Games Programming you'll be told exactly what to do.

FINDING NEW ROUTINES

Having entered all the verbs in the last routine, the computer will need some routines which will enable it to comply with the instructions, such as getting the adventurer to carry objects, for example.

The subroutine starting at Line 3Ø1Ø defines V$, N$ and I, which is a number from array R—you've already typed in this subroutine.

This short routine will enable the machine to pick out the right routine according to the value of I—the meaning of the adventurer's input.

The Spectrum has no ON ... GOTO as used in the programs for the other computers, so the program lines have to be a little different.

You already have an array, G, which contains line numbers for the location descriptions. The line numbers needed for the new routines can be added to this array.

This is why the Line 3Ø you typed in reads:

```
3Ø FOR N=1 TO 4: FOR M=1 TO 11: READ
   G(M,N): NEXT M: NEXT N
```

and why you have this line containing all the line numbers you'll need. (For a fuller explanation see page 346):

```
7Ø DATA 1Ø1Ø,115Ø,124Ø,131Ø,141Ø,
   146Ø,15ØØ,136Ø,1Ø8Ø,155Ø,311Ø
```

Now add the routine which will pick out the right routine according to the value of I:

```
5ØØ REM **FIND OPTION**
51Ø IF I=Ø THEN GOTO 52Ø
52Ø PRINT "I DON'T KNOW HOW TO□";V$:
   GOTO 37Ø
```

If the 'Check Instruction' subroutine—starting at Line 3Ø1Ø didn't find a match for V$ in R$, then I is set to zero, and Line 51Ø causes the message I DON'T KNOW HOW TO ... to be displayed. If I has any other value, Line 515 finds the correct line number in array G and executes a GOTO.

```
5ØØ REM ** FIND OPTION **
5Ø5 IF I=Ø THEN GOTO 52Ø
51Ø ON I GOTO 1Ø1Ø,115Ø,124Ø,131Ø,
   141Ø,146Ø,15ØØ,136Ø,1Ø8Ø,155Ø,311Ø
52Ø PRINT:PRINT "I DON'T KNOW HOW
   TO□";V$:GOTO 37Ø
```

Each of the numbers after the ON ... GOTO in Line 51Ø is the start of a routine. Each value of I is a different verb or group of verbs. If I=1Ø, for example, the 'kill' routine will have to be selected—it's the tenth number in the line so the routine starts at Line 155Ø.

If the 'Check Instruction' subroutine—starting at Line 3Ø1Ø—didn't find a match for V$ in R$, then I is set to Ø. In that case the ON ... GOTO in Line 51Ø will not have any effect. The message in Line 52Ø will be displayed.

TROUBLESHOOTER

- Make sure that the three pieces of DATA connected with the objects are READ into the correct array. If you try to feed string DATA into a numeric array, you will receive an error message, or you may find that a short description appears when you are expecting a long one.
- Be very careful to match the order of the pieces of DATA with the order of the arrays in the READ statement because the same problem may occur. The order is location, short description, long description.
- Do a 'dry run' on your adventure once you have entered the objects and make sure that the objects appear at the right locations.
- Use your grid when checking the objects to make sure you haven't missed any.

ACQUIRING THE OBJECTS

You already have the routine for when I=1 in your machine. I=1 when the adventurer has given a direction word and the routine is at Lines 1Ø1Ø to 1Ø6Ø.

When I=2 the adventurer has typed a 'Get' word—GET, TAKE or CARRY. This routine will allow the adventurer to pick up and keep any object that is there at the present location.

The routine looks like this:

```
114Ø REM **GET**
115Ø FOR G=1 TO NB
116Ø IF N$=B$(G, TO LEN N$) THEN GOTO
   119Ø
117Ø NEXT G
118Ø PRINT "I DON'T UNDERSTAND□";
   N$: GOTO 33Ø
119Ø IF B(G)=-1 THEN PRINT "YOU'VE
   GOT IT": GOTO 33Ø
```

```
1200 IF B(G)<>L THEN PRINT "IT ISN'T
   HERE": GOTO 330
1210 PRINT "OK": LET B(G)=-1
1220 GOTO 330
```

```
1140 REM ** GET **
1150 FOR G=1 TO NB
1160 IF N$=LEFT$(OB$(G),LEN(N$)) THEN
   1190
1170 NEXT
1180 PRINT "□I DON'T UNDERSTAND□";
   N$:GOTO 330
1190 IF OB(G)=-1 THEN PRINT "YOU'VE
   GOT IT":GOTO 330
1200 IF OB(G)<>L THEN PRINT "IT ISN'T
   HERE":GOTO 330
1210 PRINT "■OKπ":OB(G)=-1
1220 GOTO 330
```

```
1140 REM**GET**
1150 FOR G=1 TO NB
1160 IF INSTR(OB$(G),N$)=1 THEN GOTO
   1190
1170 NEXT
1180 PRINT"□I DON'T UNDERSTAND□";
   N$:GOTO330
1190 IF OB(G)=-1 THEN PRINT "YOU'VE
   GOT IT":GOTO 330
1200 IF OB(G)<>L THEN PRINT "IT ISN'T
   HERE":GOTO 330
1210 PRINT "OK":OB(G)=-1
1220 GOTO 330
```

Lines 115Ø to 117Ø search the array containing the short object descriptions—B$ in the case of the Spectrum, and OB$ in the case of the others—for the object that the adventurer has named. If the named object is found, then the program jumps to Line 119Ø. If the object is nowhere in the adventure, Line 118Ø displays the message I DON'T UNDERSTAND, followed by the name of the object the adventurer has typed in.

Assuming that the named object has been found, two checks will have to be made. Line 119Ø checks the element of the object location array—B, or OB—to see if the object is already being carried. If it *is* being carried (the value of the array element = −1) then the message YOU'VE GOT IT is displayed.

Line 12ØØ checks if the object is present by checking the location array again. If it isn't, then the program says IT ISN'T HERE. You can always change these messages of course, if they don't suit your adventure.

If the object isn't being carried and it is at the same location as the adventurer, Line 121Ø says OK and the element in the object location array is changed to −1.

DROP

The 'Drop' routine does exactly the opposite to the last one. It enables the adventure to abandon any unwanted objects.

```
1230 REM **DROP**
1240 FOR G=1 TO NB
1250 IF N$=B$(G, TO LEN N$) THEN GOTO
   1270
1260 NEXT G: PRINT "I DON'T
   UNDERSTAND□";N$: GOTO 330
1270 IF B(G)<>-1 THEN PRINT "YOU
   HAVEN'T GOT IT": GOTO 330
1280 PRINT "OK": LET B(G)=L
1290 GOTO 330
```

```
1230 REM ** DROP **
1240 FOR G=1 TO NB
1250 IF N$=LEFT$(OB$(G),LEN(N$)) THEN
   1270
1260 NEXT:PRINT "I DON'T
   UNDERSTAND□"N$:GOTO 330
1270 IF OB(G)<>-1 THEN PRINT "YOU
   HAVEN'T GOT IT":GOTO 330
1280 PRINT "OK":OB(G)=L
1290 GOTO 330
```

```
1230 REM **DROP**
1240 FOR G=1 TO NB
1250 IF INSTR(OB$(G),N$)=1 THEN 1270
1260 NEXT:PRINT"I DON'T
   UNDERSTAND□";N$:GOTO 330
1270 IF OB(G)<>-1 THEN PRINT "YOU
   HAVEN'T GOT IT":GOTO 330
1280 PRINT "OK":OB(G)=L
1290 GOTO 330
```

The routines work in a very similar way to the 'get' routines. The short description arrays are again searched—this time by Lines 124Ø to 126Ø. If the object that the adventurer has named is in the array then Line 127Ø checks if the adventurer is carrying the object. If it isn't being carried, the message YOU HAVEN'T GOT IT is displayed.

If the adventurer is carrying the object Line 128Ø says OK and the appropriate element in the object location array—OB, or B—is adjusted. It now takes the number of the current location—L—rather than −1 which meant that it was being carried.

LISTING LOOT

The forgetful adventurer will be very glad of an inventory so that all the objects that have been picked up can be listed on request. Here's a routine which will do just that:

```
1070 REM **INVENTORY**
1080 PRINT "YOU HAVE:□";: LET IN=0
1090 FOR G=1 TO NB
1100 IF B(G)=-1 THEN PRINT TAB
   10;B$(G): LET IN=IN+1
1110 NEXT G
1120 IF IN=0 THEN PRINT "ZILCH"
1130 GOTO 330
```

```
1070 REM ** INVENTORY **
1080 PRINT "◣YOU HAVE:π□";:
   IN=0
1090 FOR G=1 TO NB
1100 IF OB(G)=-1 THEN PRINT
   TAB(10)OB$(G):IN=IN+1
1110 NEXT
1120 IF IN=0 THEN PRINT "ZILCH"
1130 GOTO 330
```

```
1070 REM**INVENTORY**
1080 PRINT "YOU HAVE:□";:IN=0
1090 FOR G=1 TO NB
1100 IF OB(G)=-1 THEN PRINT
   TAB(10)OB$(G):IN=IN+1
1110 NEXT
1120 IF IN=0 THEN PRINT "ZILCH"
1130 GOTO 330
```

YOU HAVE is displayed by Line 1Ø8Ø ready for the list of items. The FOR . . . NEXT loop checks through each element of the object location array in turn. This time the important elements are the ones containing −1, meaning that the object is being carried. If the value of any of the elements is −1, then the object description is printed from the short description array. The inventory counter IN is increased by 1.

If no objects are being carried IN remains at zero and Line 112Ø displays ZILCH instead of the list of objects.

The 'Get', 'Drop' and 'Inventory' routines can be used as they stand as long as NB has been defined in an earlier routine.

Now SAVE the program ready for the final routines next time. These routines are the ones concerning the tax inspector, the brick, the lamp, finding the eyeball, ending the adventure, and, finally, the instruction describing the object of the quest.

If you RUN the program at this stage you'll find that while parts of it work, there are also some rather strange things happening. The reason for this is that there are still a number of routines which do not exist yet. If you input some words the program will try to jump to non-existent lines.

Q+A

Is it possible to use a speech synthesizer with an adventure?

You could make your adventure more interesting by programming the machine so that it will announce the messages, directions and the descriptions of the objects rather than display them on the screen.

Look in your synthesizer manual to see how the machine can be made to speak, and substitute the instructions for the PRINT statements.

INPUT will be looking at speech synthesizers in depth in a later article.



NEW IDEAS FOR SCREEN ART

Bring your graphics commands up to date and bridge the gap between drab, lifeless visuals and that bright, professional look you always dreamt your programs could achieve

Although you may know by now what most of the graphics commands do, you may not be putting them to the best use.

The colouring commands, for instance, are much more versatile than they first appear and can do a lot more than simply fill in blocks of colour. Here are a few ideas and techniques to help brighten up your graphics.

The Spectrum's PLOT and DRAW commands can be used in a wide variety of ways, but you may find that they don't always produce the effect you're after. In many cases this is due to the limitations of the high resolution graphics screen—which will not accept different colours when these are in adjacent pixels within a square on the text screen. But there are several other reasons why you might not always achieve the results you expect. What you should be doing now is to allow for these effects in your programs and even take advantage of them.

For example, if the method of shading you're using looks rather uneven, then build this into your program so it looks deliberate. And if you find that colours overflow into other areas of your drawing or you need more than two colours in a single character square then arrange your drawing so that any colour change starts on a new square. You'll still have a problem with curves but you can go a long way to lessening the effect.

So whatever you are doing, if you can't get the effect you want, be willing to adapt and use what's possible. You can then look for methods for refining your drawings later as you learn more about the machine. Here, then, are some ideas for using shading and colouring that you can use in your own programs.

SHADING THE SCREEN

If you have tried the programs in the previous graphics articles (pages 84 to 91 and 184 to 192), you should already have a good idea of some of your computer's potential for making pictures on the screen. And if you have made your own experiments using the techniques explained there, you will have discovered even more.

But now is the time to look at extending your computer graphics again, by exploring some of the more sophisticated uses for the drawing and colouring commands that you have already used. And at the same time, you will be able to extend your repertoire with a range of new pictures to call up on the screen, or use as the basis for your own ideas.

This short program, for example, PLOTs pixels in randomly generated positions. Type it in, RUN it, and see how long it takes to fill in the entire screen:

```
10 LET x=INT (RND*256)
20 LET y=INT (RND*176)
30 PLOT x,y
40 GOTO 10
```

You'll have to be patient to see a result, and this would be an impossibly slow method if you wanted to fill in completely a large area in a graphic. However, what the program does much more usefully, is to shade in an area—which it does quite quickly, if not in an even manner.

By adding one line to the program, you can create a much better effect, which you could use on its own or within a program. Type in this line, and then RUN the program again.

```
25 IF (x>35 AND x<90) AND (y>35 AND
   y<90) THEN GOTO 10
```

After only a short time you should be able to see that the computer is leaving a square totally clear from the pixels.

Line 25 checks the values of x and y, and if both of them fall between 35 and 90 then the computer jumps back to Line 10 to choose new values. So you get a blank area appearing between 35 and 90 in both directions, which forms a square at these coordinates.

You might like to use this facility in a game: you could PRINT something in a square, and then gradually fill in the rest of the screen,



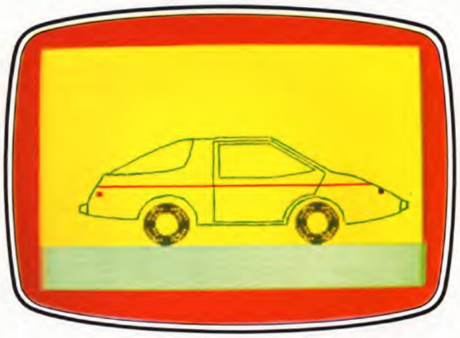
- MAKING MORE USE OF COLOUR
- SHADING AND COLOURING THE SPECTRUM'S SCREEN
- FLASHING COLOURS AND LARGE CHARACTERS ON THE COMMODORES
- USING **GCOL** TO **AND** AND **OR** COLOURS ON THE ACORNS
- MORE ON **PSET**, **PRESET** AND **COLOR** ON THE DRAGON AND TANDY

while your words are still clearly visible on the screen. The technique would be equally effective as a means of highlighting a message on the screen—such as a title page of a program or game.

The conditions in Line 25 are quite complex. On page 35, you saw how you can have more than one condition in an IF ... THEN statement by using the functions AND and OR.

These two functions have precisely the meanings that they do in ordinary English. So, in Line 25, above, x must be more than 35 AND less than 9Ø, AND y must be less than 9Ø AND more than 35 for the computer to GOTO 1Ø.

The brackets in between the two halves of the line separate the conditions for x and y (in fact the conditions are the same, but they still need to be separated by an AND). In this particular example, you could remove the brackets, but only because each of the functions is AND: if you use a mixture of ANDs and



ORs then the brackets are essential. You will see what an OR does in the following example.

You can change the area that is left untouched in several ways. For example, change the AND in between the two sets of brackets in line 25 to OR. Instead of leaving a blank square, you should now see a blank cross on the screen.

Try changing Line 25 to:

```
25 IF (x>11Ø AND x<145 AND y>4Ø AND
   y<136) OR (x>4Ø AND x<215 AND
   y>7Ø AND y<1ØØ) THEN GOTO 1Ø
```

You can leave any shape you like clear of dots but the more complicated it is, the more conditions you need in the IF ... THEN statement and the longer the computer will take checking them. Even a triangle or a circle would require a long set of conditions, with the result that the computer would take a long time to PRINT each pixel.

If you want to create your own shapes, you should restrict them to vertical and horizontal lines, as these need less checking, and so take less time to appear.

DRAWING AND COLOURING

In earlier articles, you have seen how to DRAW some quite detailed pictures on your Spectrum. And if you have LOADed any commercial programs which have a LOADing picture to introduce the program, you'll be aware that the Spectrum is capable of quite detailed pictures.

However, there is one problem for the budding graphics programmer. In the ordinary way, the Spectrum has a big limitation when it comes to DRAWing a picture and shading it in with colour. If you haven't come across this already, here is an example:

```
5 BORDER 7: PAPER 7: INK 4: CLS
1Ø FOR x=1 TO 1Ø
2Ø PRINT "■■■■■■■■■■"
3Ø NEXT x
4Ø PLOT INK 2;Ø,175
5Ø DRAW INK 2;12Ø,−8Ø
6Ø INK 1: CIRCLE 1ØØ,95,5Ø
```

This colours an area green using block graphics, then DRAWs a diagonal red line and a blue circle. Or at least, that is what *should* happen. In fact, where the lines cross the green area, all those squares they touch become converted into the new colour.

Of course, there are ways to avoid this, but most need a lengthy program to enable the Spectrum to handle the information.

A later article will look at ways to do this, but there is a simple alternative—to avoid the problem areas completely. This is what the earlier coloured graphics, like the golf course on page 185, did. There, the screen colour was specified first, and then the DRAWn lines were placed over it. And no DRAWn line was taken over an area which had already been used for another piece of DRAWing or a block graphic.

To explore this more fully, and at the same time experiment with drawing arcs, type in the following program. It draws a coloured picture of a car:

```
8Ø BORDER 2: PAPER 6: INK Ø: CLS
9Ø FOR n=8 TO 15
1ØØ CIRCLE 8Ø,47,n: CIRCLE 18Ø,47,n
11Ø NEXT n
12Ø PLOT 62,51: DRAW 38,−5,−PI
13Ø DRAW 6Ø,Ø: DRAW 4Ø,Ø,−PI
2ØØ FOR n=1 TO 1Ø: READ a,b,c: DRAW
   a,b,c: NEXT n
21Ø PLOT 112,48: DRAW 45,Ø: DRAW
   22,2Ø,−PI/2
22Ø DRAW 7,15: DRAW −35,23: DRAW
   −4Ø,Ø: DRAW Ø,−58
23Ø PLOT 115,1Ø4: DRAW 34,Ø: DRAW
   3Ø,−19
24Ø DRAW −64,−5,−.25: DRAW Ø,24
25Ø PLOT 4Ø,83: DRAW 52,5,.3: DRAW
   6,2Ø,.3
26Ø PLOT 3Ø,55: DRAW −5,1: DRAW Ø,5:
   DRAW 5,1
27Ø PLOT 24Ø,56: DRAW 5,1: DRAW Ø,5:
   DRAW −5,1
```



```
280 PLOT 36,75: DRAW INK 2; 184,Ø
290 PRINT OVER 1; INK 2;AT 13,4;"▛";AT
    13,28;"▛"
300 FOR n=31 TO Ø STEP -1: PLOT Ø,n:
    DRAW INK 4;255,Ø: NEXT n
500 DATA 4Ø,8,.2,Ø,1Ø,Ø,-3Ø,15,
    .2,-2Ø,5,.2
510 DATA -4Ø,25,Ø,-6Ø,-2,.1,-5Ø,
    -25,.2
520 DATA -1Ø,-2Ø,-.25,Ø,-8,Ø,31,
    -3,.2
```

Lines 9Ø to 11Ø draw a series of circles, each one pixel larger in radius than the previous one, which form the wheels of the car. There is a dotted effect on the tyres, which is caused by the Spectrum not being able to draw true circles on the square grid of pixels. This means that there is a slightly stepped effect to the curve and some of the pixels are not part of any of the circles, and so are not PLOTted.

Normally this might be a nuisance, but here it is actually an advantage, since it gives the wheels a more realistic appearance. One of the keys to producing good pictures is to make use of rather than be troubled by, the characteristics of the medium.

Line 12Ø PLOTs a pixel at 62,51 which is the starting position for the lines which form the car. Using the FOR ... NEXT loop in Line 2ØØ, and Lines 12Ø and 13Ø, the Spectrum READS the information which it needs to DRAW the outline of the car. The DATA for the FOR ... NEXT loop is held in Lines 5ØØ to 52Ø. Notice that all the curves are held in the form x, y, z, which is the Spectrum command for a curved line (an arc), where z specifies the curvature.

The first two numbers are the same as usual: the number of pixels that you want the last pixel in the new line to be above and to the right of the present PLOT position. The third number sets the angle of the curve. The command to DRAW the curved bonnet of the car is DRAW 38, -5, -PI. Try changing the last number (PI is roughly 3·14) and see how curved a bonnet the computer will DRAW.

The details within the outline are sections: the door and its window in Lines 21Ø to 24Ø, the rear window in Line 25Ø, the bumpers in Lines 26Ø to 27Ø and the 'go-faster' stripe in Line 28Ø. Lines 29Ø and 3ØØ simply add a touch of colour to the picture: Line 29Ø puts in the indicators (in red), and Line 3ØØ fills in the grass on which the car is standing.

By careful placing of the grass and car, you can avoid the problem of having two colours in each character square. Look at the grass beneath the car in the program above. The car has been positioned with its wheels reaching down to the bottom of a character space, so that the grass can start on the very next pixel without changing the colour of the tyres. As you can imagine, this means that you have to plan the position of each element very carefully.

Notice, though, that this is not always possible. The go-faster stripe, for example, is coloured red, and small parts of the outline of the car in the same square have been changed to red. With care, you can often avoid having colour clashes in areas where it would show up badly.

To practise your DRAWing, you might like to change parts of the car. That way you know what the changes you are making to the program actually do and you can get used to DRAWing and being able to imagine the finished result of your pictures without having to DRAW them completely from the beginning.

No direct graphics commands are available on the Commodore 64 and this means you have to use something like the Simons' BASIC cartridge which offers these facilities. The majority of the graphics related commands available on Commodore 64s fitted with this accessory have been explained already (see pages 84 to 91 and 184 to 192). Several others remain and these relate to the use of text with graphics, and to screen manipulation.

FLASHING

Flashing screen and prompt displays serve an important function in certain types of program—especially in games. Neither is particularly difficult to provide within a normal BASIC program, (see page 49) but Simons' BASIC provides a simple set of commands for this: FLASH, BFLASH and OFF. These commands cannot be used with high-res and multicolour graphics modes.

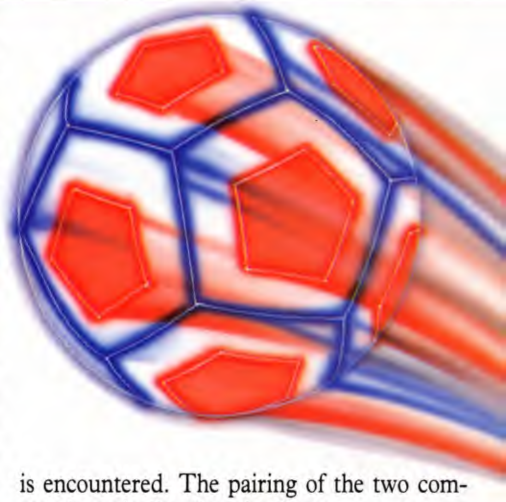
The FLASH command alternates the screen display between normal and reverse field colours. It is used in the form:

```
FLASH Ø,1Ø
```

where the two numbers following the command specify the colour and speed. The first value can range between Ø and 15 to encompass the normal range of colours available on the 64 (details are in your manual). Black (colour set value Ø) has been used here.

The second value controls the rate at which the FLASHing takes place. The default figure if you leave this value off is a FLASH once every four seconds. But a figure in the range 1 to 255 can be entered to control the flashing speed up to this maximum. Each unit corresponds to one of the system timing units called a 'jiffy', about one-sixtieth of a second.

FLASHing continues until the command OFF is encountered. The pairing of the two commands typically takes the form:



```
5 PRINT "🗋"
10 PRINT AT(12,1Ø) "THIS IS FLASHING"
20 FLASH Ø,1
30 PAUSE 3
40 OFF
50 PRINT AT(12,1Ø) "□□□SO IS THIS
   □□□"
60 FLASH Ø,5
70 PAUSE 3
80 OFF
90 GOTO 1Ø
```

Try changing the values in Lines 2Ø and 6Ø to alter colours and the FLASHing rate.

BFLASH is used to flash the screen border area. It takes the form:

```
BFLASH 1Ø,Ø,1
```

The first figure after the command regulates the flashing rate and again the values can range from 1 to 255 to give a maximum time of about four seconds.

The second and third values relate to the border colours—again using the normal range of colour values for black and white.

To turn the border flashing off simply use:

```
BFLASH Ø
```

CHARACTER COMMANDS

Text on a graphics screen can be useful for annotation or labelling, serving a very necessary function on graphics produced for educational or business purposes.

Simons' BASIC has several commands of this type. CHAR enables you to display text characters one by one on a high-res or multicolour graphics screen. It is used in the form:

```
CHAR 1Ø,5Ø,65,1,4
```

The first pair of figures give the character position on the screen in standard X and Y pixel order. Next is the POKE code of the letter



you wish to display—note that these are the screen code values and not ASCII. Next is the plot type figure which is used in the same way as it is with other Simons' BASIC commands. 1 here indicates 'plot a dot on the screen'. The final figure designates the screen character size. The value can range from 1 to 8, giving character heights from 8 pixels (value 1—1—normal size) to a maximum of 64 pixels (value 8—eight times normal size).

This program shows CHAR in use to display the alphabet in the centre of the screen, which could be used to teach a child the alphabet.

```
10 HIRES Ø,1
20 FOR N=1 TO 26
30 CHAR 15Ø,8Ø,N,1,4
40 PAUSE 1
5Ø CHAR 15Ø,8Ø,N,Ø,4
6Ø NEXT: GOTO 2Ø
```

Note that Line 5Ø is used with a plot value of Ø to wipe out the previous entry. See what happens if it is left out (place a REM immediately after the line number).

Although the CHAR command can be used to add letters one at a time, another command—TEXT—is better for displaying character strings. It takes the form:

```
TEXT 1Ø,1Ø,"ANNOTATION",1,4,4
```

The first pair of figures once again designate the X and Y start position of the character string which follows in quotes. The next figure—1—is the familiar plot type value. The first of the next pair of figures designates the character height in the same way as before. The next and final figure gives the pixel spacing between each letter.

By using an embedded code within the character string you can specify whether text is to be displayed in upper case or in lower case.

For upper case, immediately after the first quote mark press the CTRL key and A key simultaneously. A reverse-field A is displayed. Then complete the string with the chosen message, closing with quote marks.

For lower case displays hold down CTRL and B keys instead. This displays a reverse-field B.

Upper and lower case characters may be mixed by preceding a letter or group of letters with the appropriate reverse-field symbol.

Both these commands will appear again when we look at how to program graphs and charts.

CSET is a graphics related command which is useful for switching between uppercase/graphics mode and upper/lowercase mode. It selects the first of these when followed by the value Ø, and the second when followed by 1. But by following the command with the value 2 you can recall the last graphics screen which was shown. The following program makes use of most of the graphics commands that have been examined to date.

Here's a program you could use as a basis for teaching pre-school children. It makes use of many of the graphics.

ALPHABET PICTURES

The program draws pictures for the first three letters of the alphabet when one of these is selected after the opening prompt. All remaining letters are shown briefly before the program passes back to the prompt. As an exercise, you could embellish this program to provide your own pictures and picture-making subroutines in addition to the three provided.

A Vic 20 version of this program is possible using the Super Expander cartridge and the listing for this is shown also. This too may be altered to suit your own requirements.






```
1Ø HIRES 1,Ø:MULTI Ø,5,7:COLOUR 6,2
15 BLOCK Ø,3,16Ø,8Ø,1
2Ø TEXT 6,6Ø,"ENTER A LETTER OF
   THE",2,1,7
3Ø TEXT 5Ø,11Ø,"ALPHABET",3,4,8
4Ø POKE 198,Ø
5Ø GET A$:IF A$<"A" OR A$>"Z"
   THEN 5Ø
55 TEXT 75,5,A$,3,5,7:FOR Z=1 TO
   5ØØ:NEXT Z
```



```
60 A=ASC(A$)-64:ON A GOTO 1000,
   2000,3000
65 TEXT 75,5,A$,1,5,7
99 GOTO50
1000 HIRES 0,1:MULTI 5,13,2:COLOUR 6,7
1005 CIRCLE 80,100,40,40,2
1010 PAINT 70,70,3
1015 FORZ=1TO50STEP.3:PLOT(65+Z*.5)
   +RND(1)*25,80+RND(1)*40,RND(1)
   *3:NEXT
1020 FOR Z=1 TO 3:ARC 75-Z,70,0,90,
   10,10,30,Z:NEXT Z
1030 ARC 95,55,0,360,65,15,8,3:PAINT
   95,55,1
1035 ARC 65,55,0,360,65,15,5,3:PAINT
   65,55,2
1040 A$="A":B$="A□P□P□L□E":
   GOTO 9000
2000 HIRES 0,1:MULTI 2,5,6:COLOUR 7,7
2010 FOR Z=43 TO 16 STEP -5:CIRCLE
   80,100,50-Z,58,3
2015 PAINT 83-(50-Z),100,INT(RND(1)
   *3)+1:NEXT
2020 A$="B":B$="□B□A□L□L":
   GOTO 9000
3000 HIRES 0,1:MULTI 0,1,4:COLOUR 6,13
3010 CIRCLE 80,40,13,25,3
3020 CIRCLE 80,110,20,45,2
3030 FOR Z=1TO3:ARC 53-Z,140-Z,100,
   250,15,15,10,Z:NEXT
3040 ARC 66,20,0,360,90,5,15,1:PAINT
   66,20,3
3050 ARC 93,20,0,360,90,5,15,1:PAINT
   93,20,3
3060 PAINT 80,100,1:ARC 75,35,0,360,
   90,4,3,2
3065 ARC 85,35,0,360,90,4,3,2:PAINT
   80,50,1
3070 FORZ=2TO3:LINE 60,60-Z*3,80,50,
   Z:LINE 100,60-Z*3,80,50,Z:NEXT
3080 CIRCLE 80,48,2,2,2
3099 A$="C":B$="□□C□A□T"
9000 TEXT 10,10,A$,3,5,8
9005 TEXT 140,10,A$,3,5,8
9010 TEXT 40,160,B$,1,2,10
9099 PAUSE 5:GOTO 10
```

```
10 GRAPHIC 0:COLOR 1,1,6,6
20 PRINT"■■■■■■■ENTER
   A LETTER OF THE"TAB(95)
   "■ALPHABET":POKE 198,0
30 GET A$:IF A$<"A" OR A$>"Z"
   THEN30
35 PRINT"■■"TAB(10)A$:A=ASC(A$)
   -64:FOR Z=1 TO 500:NEXT Z
40 GRAPHIC 1:ON A GOSUB 1000,2000,
   3000:GOTO 10
1000 COLOR 0,3,5,2:CIRCLE 1,500,500,
   150,145
1005 PAINT 3,500,500
1010 DRAW 1,500,370 TO 470,260:FOR
   Z=0 TO 30 STEP 3
1020 DRAW 2,485,380 TO 520+Z,
   300-Z TO 620+Z,320:NEXT Z
1030 A$="A":B$="A□P□P□L□E":
   GOSUB 9000
1099 RETURN
2000 SCNCLR:COLOR 0,2,4,6
2010 FOR Z=10 TO 250 STEP 80
2020 CIRCLE 1,500,500,Z,Z:NEXT
2022 CIRCLE2,500,300,200,200,5,48
2025 PAINT 3,500,400:PAINT 2,500,300
2027 PAINT 3,500,700:PAINT 2,500,600
2030 A$="B":B$="□B□A□L□L":
   GOSUB 9000
2099 RETURN
3000 SCNCLR:COLOR6,0,3,1
3005 CIRCLE2,500,300,90,100
3010 CIRCLE2,500,600,100,200
3015 PAINT 3,500,300:PAINT3,500,600
3020 DRAW 1,400,250 TO 380,170 TO
   430,240
3025 DRAW 1,600,250 TO 620,170 TO
   570,240
3030 FORZ=0TO8:CIRCLE 1,300,700+Z,
   100,50,0,40
3040 POINT 1,460+SIN(Z)*10,300+
   COS(Z)*10,540+SIN(Z)*10,300+
   COS(Z)*10:NEXT
3045 FOR Z=0TO30STEP30:DRAW 2,400,
   350+ZTO500,370TO600,350+Z:NEXT
3050 A$="C":B$="□□C□A□T":
   GOSUB 9000
3099 RETURN
9000 CHAR 1,9,A$:CHAR 17,5,B$
9010 FOR Z=1TO4000:NEXT:RETURN
```





If you have tried the graphics routines already covered in *INPUT*, you have seen how to draw shapes on the screen, and how to add to the scope of your pictures with colour. But there is much more you can do using the Acorn micros' sophisticated colour potential.

Before you can attempt to draw any sort of graphics on the Acorn computers you first have to put the computer into one of the graphics modes. There are five modes you can use—modes Ø, 1, 2, 4 and 5—and each has a different resolution and a different number of colours. Only mode 2 supports the full range of 16 colours (or rather eight colours and eight flashing combinations) and since this article is about colour this is the mode to use. So type MODE 2 and press RETURN.

FOREGROUND AND BACKGROUND

The colour command for graphics is GCOL, and it works in much the same way as the COLOUR command for text. If you want to specify a *foreground* colour just use the logical colour number of the colour you want (see the manual for a list of colours). For instance, COLOUR 1 gives red text and GCOLØ,2 gives green graphics. To prove it type in those last two commands then type:

```
CLS:PRINT "TEXT":DRAW 1ØØØ,1ØØØ
```

Note that COLOUR only works with text and GCOL only works with graphics.

To change the *background* colour use the same commands but add 128 to the colour number. So to change the graphics background to yellow (logical colour 3) type GCOLØ,131. If you are wondering why there is no change, try entering CLG. This clears the graphics screen to the new background colour. Exactly the same applies to the text screen; COLOUR 132 followed by CLS clears the text screen to blue. Try typing CLG followed by CLS a few times to swop the screen colour between yellow and blue.

SIXTEEN COLOURS

You can select any of the sixteen colour effects for the foreground by entering a number between Ø and 15, and any colour for the background by entering a number from 128 to 143. Enter and RUN the next program to see all 16 colours:

```
1Ø MODE 2
2Ø FOR C=Ø TO 15
3Ø COLOUR C
4Ø PRINT TAB(1Ø) "X"
5Ø NEXT
```

If the flashing colours start to irritate you then change the 15 in Line 2Ø to 7. Also try changing MODE 2 to any of the other modes. The size of the image will change and you'll see fewer colours too.

The last program printed a text character and so used the COLOUR command, but one of the features of BBC BASIC is that you can print text at the graphics cursor, as long as you use VDU 5 first. Type in these three Lines to see fewer colours too.

```
15 VDU 5
3Ø GCOLØ,C
6Ø VDU 4
```

The 'X' is now treated as graphics. This is quite useful because it means text can be positioned very accurately by MOVEing to a particular position rather than using PRINT TAB. In MODE 2 you can MOVE to 160 × 256 positions whereas it is only possible to PRINT TAB at 20 × 32. Another consequence of treating the text as graphics is that GCOL is used to colour it and as you'll see in a moment, GCOL is a lot more versatile than COLOUR. But to realize the full benefits of GCOL, you must understand the use of logical operators. Exactly how they work will be described in a little while, but if you've forgotten the meaning of EOR or any of the other logical operators then have a look at the article on pages 284 to 288.

The logical operators are extremely useful in graphics programming, so let us look at



them in detail. For clarity, the example uses a circle drawing routine, which gives a larger image than the single character UDG. Type NEW then enter and RUN the next program:

```
1Ø MODE 2
2Ø VDU 29,64Ø;512;
3Ø R=4ØØ
4Ø MOVE Ø,Ø
5Ø GCOL Ø,13Ø:CLG
6Ø FOR T=Ø TO 2.Ø1*PI STEP PI/14
9Ø MOVE Ø,Ø
11Ø PLOT 85,R*COS T,R*SIN T
12Ø NEXT
```

This program places the origin at the centre of the screen (Line 2Ø), moves the cursor there and clears the screen to a green background (Line 5Ø). Lines 6Ø to 12Ø PLOT a circle in white (the default colour).

Now enter a new line, Line 1ØØ, to change the colour of the circle to any of the 16 colour effects, except green (green is the background colour, so any shape plotted in green will not appear). Try red first:

```
1ØØ GCOL Ø,1
```

When you RUN the program, the circle appears red—the colour specified by GCOL Ø. Now see what happens when you change Line 1ØØ to GCOL 1,1. The colour specified is no longer red (colour 1), but the colour produced by ORing red with green—the colour already on the screen. This new colour is yellow. To make sense of this you really have to think in binary. Red OR green is colour 1 OR colour 2. In binary, this is Ø1 OR 1Ø—giving a result after ORing of 11, which in decimal is 3. From your manual, you will see that colour 3 is yellow.

Changing Line 1ØØ to GCOL 2,1 gives a black circle, because the '2' specifies AND. 1 AND 2 in binary is Ø1 AND 1Ø, which gives the result Ø—black.

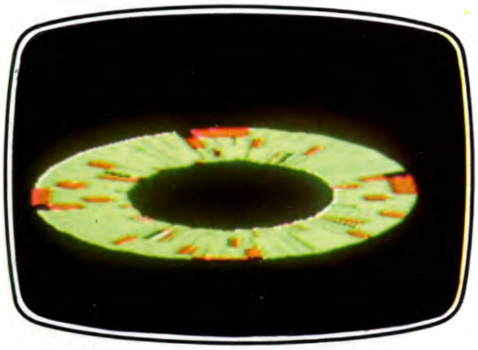
GCOL 3,1 specifies red EOR green and again this happens to be yellow because Ø1 EOR 1Ø is 11. EOR doesn't always have the same effect as OR of course. If the circle was first plotted in yellow then yellow OR green is yellow (11 OR 1Ø is 11) but yellow EOR green is red (11 EOR 1Ø is Ø1).

The last logical operator (NOT) is specified by GCOL 4, which has the effect of inverting the colour already on the screen (see the manual for a complete list of logical colours).

A SECOND IMAGE

One of the main advantages of the logical operators is that they allow you not only to











change colours, but also to define more than one image and to select which ones appear at any one time. Change the program above by deleting Line 5Ø and adding Lines 7Ø, 8Ø and 1ØØ, then add Lines 13Ø to 16Ø to plot an inverted triangle. Here is the whole program with all the changes made:

```
10 MODE2
20 VDU 29,640;512;
30 R=400
40 MOVE 0,0
60 FOR T=0 TO 2.01*PI STEP PI/14
70 C=C+1
80 IF C>7 THEN C=1
90 MOVE 0,0
100 GCOL 0,C
110 PLOT 85,R*COS T,R*SIN T
120 NEXT
130 GCOL 0,3
140 MOVE -200,200
150 MOVE 0,-200
160 PLOT 85,200,200
```

When this program is RUN, you should see a yellow triangle at the centre of a circle with coloured segments. The colours of the segments are specified at Lines 7Ø and 8Ø, then called at Line 1ØØ. The triangle appears yellow (specified at Line 13Ø), and it blots out the circle where they meet. This is because the GCOL Ø statement plots the colour specified, regardless of what was on the screen already. Notice that if the routine to plot the triangle occurred at the start of the program, then the triangle would have been blotted out by the circle, which itself is plotted with a GCOL Ø.

Now try using GCOL 2 in Lines 1ØØ and 13Ø. Remember this ANDs colours with the ones underneath. In this case the background is black—colour Ø. Since anything ANDed with Ø is Ø, every colour in the circle appears as black and the triangle is black too. So you don't see a thing!

If you set Line 1ØØ to GCOL Ø,3 and Line 13Ø to GCOL 2,3 you might expect to see the coloured circle without the triangle. In fact, both images appear. Where they overprint, the colours are different because the yellow of the triangle ANDed with each colour of the circle is sometimes a different colour. From this image you can see that yellow AND red is red, while yellow AND green is green, but yellow AND blue is black, yellow AND magenta is red, yellow AND cyan is green and yellow AND white is yellow. Notice that the triangle appears dark, because it contains none of the lighter shades—blue, cyan or white.



PRACTICAL USES

Although this might seem rather theoretical and of no practical value, it is in fact a very useful aid to creating all sorts of animated graphics. By selecting the right logical operator, you can obscure an image or plot it in other colours. Treating an image in this way can form the basis of an action-packed visual display. Enter the next few lines and RUN the program to see how the circle of coloured segments can be turned into an attractive spinning wheel.

```
170 FOR T=0 TO 130000
180 X=T MOD 7+1
190 FOR P=1 TO 10
200 VDU 19,P,X;0;
210 X=X+1
220 IF X>7 THEN X=1
230 NEXT
240 A=INKEY(4)
250 NEXT
```

Lines 17Ø and 18Ø select a value of X between 1 and 7. The FOR ... NEXT loop from Line 19Ø to 23Ø changes ten colours of the circle (Line 19Ø) to each of these X-values. After a short delay (Line 24Ø), the ten colours are changed to the next X-value, and so on. This has the effect of shifting the colour of segments of the circle in one direction, giving the impression of spinning.

Here is a program that combines the use of GCOL Ø and GCOL 3 with the spinning effect to give an attractive display.

```
10 MODE 2
20 VDU23;8202;0;0;0;
30 VDU 29,640;200;
40 MOVE 0,0
50 N=0:R=200:R2=600
60 FOR T=20*PI TO 9.99*PI STEP -.1
70 IF T<12*PI THEN GCOL 0,0:GOTO 110
80 GCOL 3,N:N=(N+1) MOD 15
90 IF T>18*PI THEN 110
100 R=R-.5:R2=R2-1.5
110 MOVE 0,0
120 PLOT 85,R2*COS T,R*SIN T
130 NEXT
140 VDU 20
150 E=130
160 P=0: C=7
170 REPEAT
180 C=C+1: IF C=9 THEN C=7
190 FOR L=1 TO 15
200 E=E-.4
210 FOR N=0 TO E
220 NEXT
230 VDU 19,L,C;0;
240 NEXT
250 IF E<5 THEN E=4.9:P=P+1: PRINT
260 UNTIL P>65
270 G=INKEY(50):GOTO 10
```

To give a perspective view of a 'flying saucer', an ellipse is plotted, instead of the circle used before. Line 5Ø sets the value of the first colour and the axes of the ellipse. Line 7Ø ensures that the ellipse is plotted with a hole at the centre, because it plots black on black. Line 8Ø specifies that a colour between 1 and 15 should be exclusively ORed with colours on the screen. Line 9Ø ensures that the circumference of the ellipse is continuous—the axes are reduced only after the first complete revolution. Line 1ØØ sets the rate at which the axes decrease, and Line 12Ø PLOTs the colours.

The rest of the program changes logical colours to give the effect of rotation. Line 18Ø selects white and flashing white/black, and Lines 21Ø and 22Ø quickens the speed of rotation. The final action is achieved by Line 25Ø. From a multicoloured ring, the image becomes a pulsating black and white vehicle. Try deleting the IF statement from Line 18Ø, and change Line 255:

```
180 FOR C=7 TO 15
255 NEXT
```

or change the values to experiment with different coloured effects. Also, change the value of the GCOL statement at Line 8Ø to see the effect of other logical operators. Some effects are obviously better than others!

You have already seen how to use the PSET command to plot sine and cosine curves, and circles on pages 241 and 242, and it has often been used in other programs. Now is the time to explore exactly the way PSET, and its related functions, PRESET, PCLS, and COLOR, work.

THE PSET COMMAND

PSET simply sets the smallest unit of graphics in any of the PMODEs to the colour you've specified—in PMODE 4 one pixel is set, in PMODEs 2 and 3 a pair of pixels is set, and in PMODEs Ø and 1 four pixels are set. Don't confuse PSET used in this way with the PSET you've used previously with the LINE command—see page 9Ø.

Try typing in this program and RUNning it:

```
1Ø PMODEØ,1
2Ø PCLS
3Ø SCREEN1,1
4Ø FOR X=Ø TO 255
5Ø Z=(X-127)/1Ø
6Ø Y=95-15Ø*Z/(1+Z*Z)
7Ø IF Y<Ø OR Y>191 THEN 9Ø
8Ø PSET (X,Y,5)
9Ø NEXT
1ØØ GOTO1ØØ
```

The program plots a graph in the coarsest of the two-colour modes—you could just as well have chosen to draw in PMODE4, or PMODE2. The graph is one of an obscure mathematical function, chosen simply because it produces quite a nice shape!

When using PSET you must tell the machine where you want to set the pixel, or pixel block, and in which colour. The colours available will depend on the graphics mode and colour set you've chosen.

The screen colour will be the lowest numbered colour in the colour set, unless you choose to change it—see later on. In a two-colour mode, when PSETting you must specify the higher numbered colour or you won't see a thing. In a four-colour mode the situation is slightly different. This time you can choose between the three highest numbered colours.

If you have a closer look at the program you'll see that the Black and Buff colour set has been specified by Line 3Ø. Line 8Ø sets the pixel at X,Y—worked out by the FOR ... in Line 4Ø, and the equations in Lines 5Ø and 6Ø. The last figure is the colour of the pixel—in this case colour 5, Buff.

FOUR-COLOUR MODES

Try changing Line 1Ø so that it reads:

```
1Ø PMODE1,1
```

Now RUN the program. Don't worry if nothing happens, because it shouldn't—yet! You are, in fact, PSETting Buff pixels on the Buff background. You'll have to change the last figure after PSET in Line 8Ø to select a different colour before anything appears. In this colour set you have the choice of cyan (6), magenta (7) and orange (8).

Try changing the figure 5 in Line 8Ø to 6, 7 or 8. You could also try changing the colour set by changing Line 3Ø so that it reads:

```
3Ø SCREEN1,Ø
```

You can now use colours numbered from 1 to 4, although 1 will PSET the same colour as the screen. To confuse the issue slightly, if you choose to use a figure from 5 to 8 with colour set Ø you won't get an error message. The machine will automatically subtract four from the colour number.

THE PRESET COMMAND

PRESET is the reverse of PSET—broadly, PRESET means 'switch off the pixel or pixel block'. More particularly, it means set the pixel, or the pixel block to the background colour—but more about foreground and background colours later.

To see PRESET at work, RUN the program as it stands. Next alter the program so that Line 8Ø reads:

```
8Ø PRESET(X,Y)
```

Don't RUN the program, because it'll wipe out the shape on the high resolution screen, but type GOTO 3Ø instead. You'll see each of the pixels—or pixel blocks—disappear one-by-one as they are changed to the background colour.

SCREEN COLOUR

You can change the screen colour to any of those in the chosen colour set. All you need do is to add a number to the end of PCLS in Line 2Ø.

With Line 3Ø selecting SCREEN1,1, try using PCLS6, PCLS7 or PCLS8 in Line 2Ø. PCLS5 will have exactly the same effect as PCLS because both will clear the screen to buff.

Once you have finished experimenting put Line 2Ø back as it was:

```
2Ø PCLS
```

FOREGROUND AND BACKGROUND

The LINE command, as you saw on pages 90 to 91, uses PSET and PRESET in a different way from the one which has just been covered.

PSET, when used with LINE, tells the computer to draw in the foreground colour, whilst PRESET tells the computer to draw in the background colour.

When you first switch on the Dragon or Tandy, the foreground colour is the highest numbered colour in the colour set—see your manual for colour numbers—and the background colour is the lowest numbered colour.

But, suppose you are in a four-colour mode and want to draw a line in one of the other colours in the colour set. This is no problem, since there is a BASIC command which allows you to specify which colour is the foreground colour, and which the background.

Type in this program and you'll see how the COLOR command works:

```
1Ø PMODE3,1
2Ø PCLS
3Ø SCREEN 1,Ø
4Ø FOR K=1 TO 4
5Ø FOR J=1 TO 4
6Ø COLOR K,J
7Ø LINE(Ø,5Ø*K+1Ø*J-55)-(255,
   5Ø*K+1Ø*J-55),PSET
8Ø LINE(Ø,5Ø*K+1Ø*J-52)-(255,
   5Ø*K+1Ø*J-52),PRESET
9Ø NEXT J,K
1ØØ GOTO 1ØØ
```

The program draws pairs of parallel lines, one in the foreground colour, and one in the background colour—you can't see all of the lines, because when either the foreground or background colour is set to green, it's the same as the screen colour.

The COLOR command in Line 6Ø provides the variation—notice that the LINE commands in Line 8Ø are *not* altered during the program, except in the end coordinates.

The COLOR command works like this: the first number after COLOR is the foreground colour, and the second is the background. The COLOR command must have both numbers, so if you don't want to change both the foreground and background colours you must specify one of them as the existing value. There's nothing to stop you having both foreground and background colours the same —although there's not much point in this.

Dragon and Tandy owners often get confused by the words foreground and background. The background isn't necessarily the screen colour. If you've used PCLS with no number in your program, then the screen colour will be the background colour. If you've specified a number with PCLS, though, the screen colour may not be.

Similarly, the foreground colour is almost certainly not the colour you are drawing in. The only graphics command which uses the concepts of foreground and background is LINE, and as you've seen you can choose to draw in either the foreground or background colours.

AVOIDING PITFALLS

- USING ACCURATE PROMPTS
- BUILDING IN ERROR TRAPS
- EXCLUDING INVALID VALUES
- ERROR INDICATION
- BOMBPROOFING

Even when a program is completely debugged errors can still occur. They result from the way a program is used—or abused—and it is these which are examined here

The key word is *use*. No matter how well a program may be from a solely technical standpoint, if it proves difficult, misleading or confusing to use, someone, somewhere, is going to have problems. And if anyone experiences problems using a computer, the chances are that an error of some kind will result.

The secret really is making a program 'user friendly' so that every stage of the program is adequately heralded and every action by the user is done knowingly.

This means providing plenty of screen displays and prompts—and adequately safeguarding both the user and the program against simple errors.

If possible, it's wise to build into your program protection against every conceivable form of accidental entry. Wrong keypresses, double presses, illegal entries, impossible values—all spell doom for certain types of program unless prevented. And to do this you can incorporate several types of error checking and input validation routines within your programs. Many of the programs so far used in *INPUT* have indeed made use of these.

PROMPTING

Accurate prompts play a large part in helping users understand quite how to respond to the options available when, say, presented with a menu.

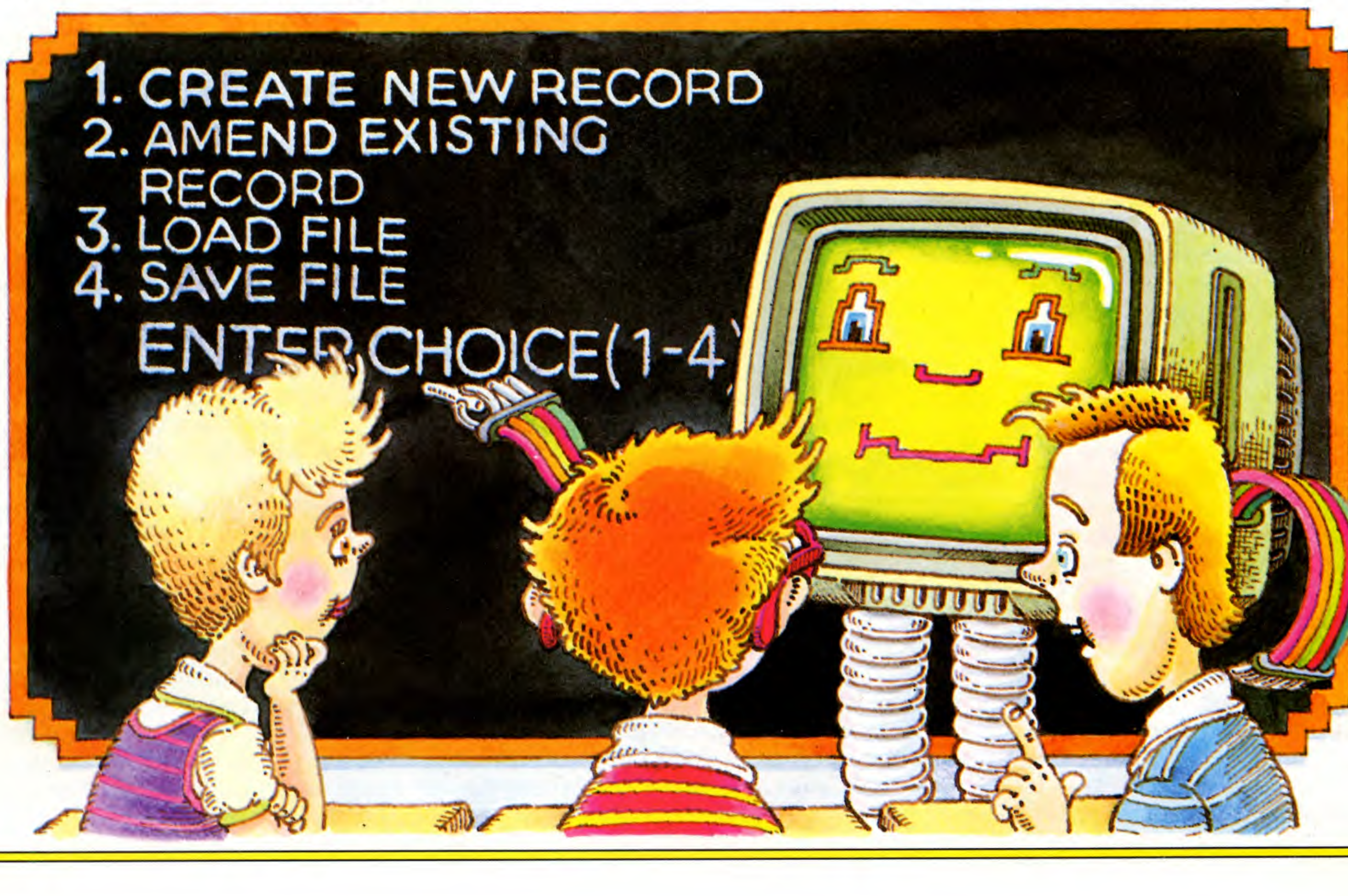
Suppose, for example, a menu displayed the following options, typical of the program entry point of a simple database:

1. Create new record
2. Amend existing record
3. Load file
4. Save file ... and so on.

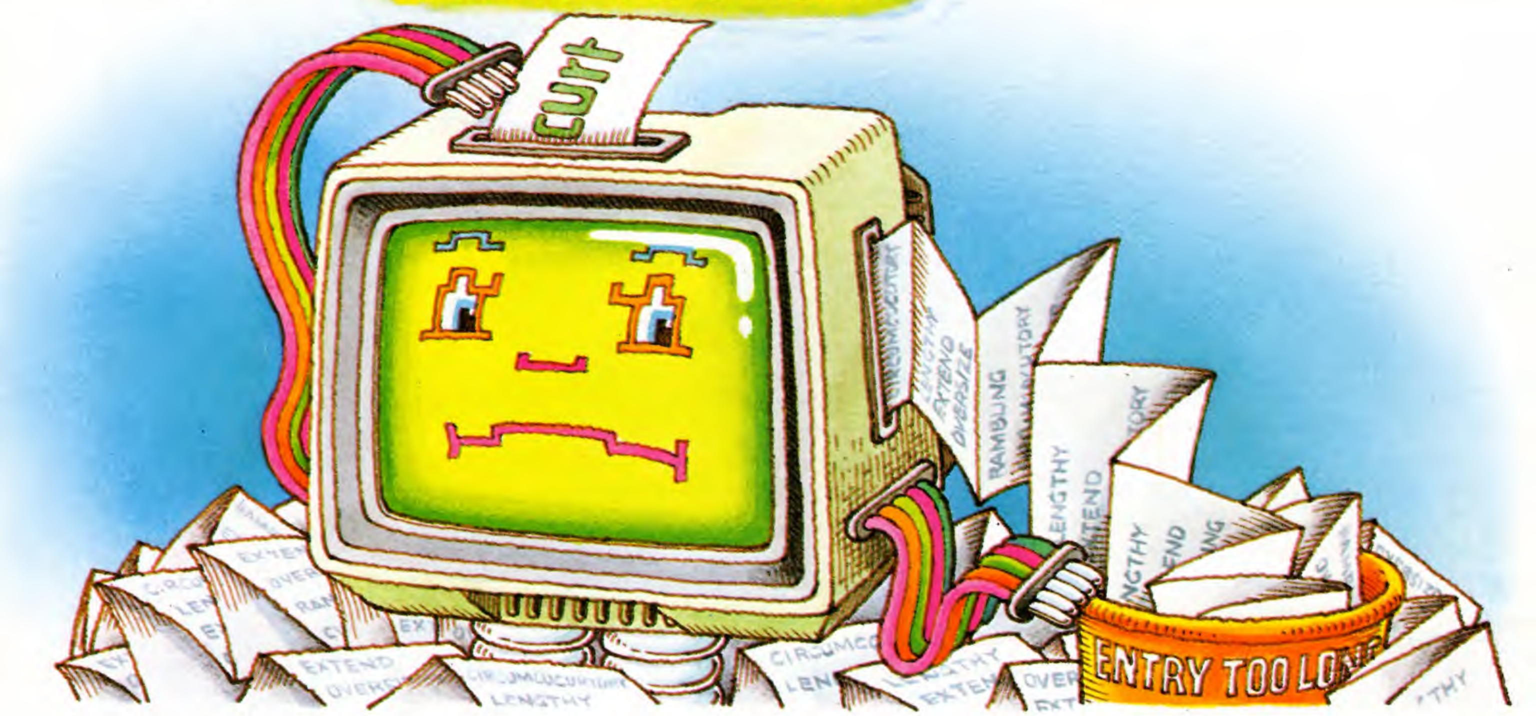
Now if you follow this with the prompt 'SELECT OPTION' or 'MAKE YOUR CHOICE', the user is left a little in the dark about how to go about what to do next. Does he or she enter the option number—or type in one or more letters of the entry?

Clearly, a much more satisfactory prompt is 'ENTER CHOICE (1 – 4)' or 'PRESS KEY 1 – 4 TO INDICATE CHOICE'.

Similarly, in prompts which require a simple 'yes' or 'no' response, indicate this in the display. So rather than use something like 'ANOTHER GO?'—where 'yes' could be indicated by a non-specific keypress, or by pressing the joystick 'fire' button, or by pressing Y—spell out the options. 'PRESS FIRE



ENTRY TOO LONG



BUTTON FOR ANOTHER GAME' and 'ANOTHER GO (Y/N)?' are both much more direct and so give clear direction on how to proceed.

Even the temporarily confusing 'PRESS ANY KEY TO CONTINUE' prompt is better in this respect. But an improvement on this much-used favourite is to specify a key—just in case the user decides to press something like STOP, BREAK or ESCAPE and so possibly exit the program! There can be no confusion over a prompt like 'PRESS C TO CONTINUE'.

On some computers, the required keypress can be highlighted by printing an inverse of the character in the prompt—often a very neat alternative.

Regardless of the types of prompt and keystroke options, it is good programming sense to disable any key which would cause the program to halt, and use routines which exclude all 'impossible' INPUTs (as shown on) page 377). To recap, for a simple 'yes or no' option you could use something like:

```
90 LET A$=INKEY$
92 IF A$="" THEN GOTO 90
95 IF A$<>"Y" AND A$<>"N" THEN
   GOTO 90
```

```
90 GET A$
95 IF A$<>"Y" AND A$<>"N" THEN 90
```

```
90 A$=GET$
95 IF A$<>"Y" AND A$<>"N" THEN 90
```

```
90 A$=INKEY$
95 IF A$<>"Y" AND A$<>"N" THEN 90
```

Much more sophisticated routines can be used to restrict input to certain value ranges (see page 319).

Another way of lessening the burden on the user is to restrict the actual amount of information which has to be input at any one given instance. If a single keypress can do the job, use it! And to avoid any confusion, use the same type of prompt/response throughout a menu-driven program. So if selection from the opening menu is made by keying in a single number, try to use the same system for all other menus which follow.

While on this point, use the same conventions for each and every menu or options list. If the third option is 'SAVE' on one menu but 'QUIT' on another, someone may not be too happy with your program in the future...

Where data input is required, the same rule can apply, and particularly when a lengthy sequence is involved. A multiple INPUT prompt is fine where data is restricted to a set and very simple pattern—a name and address file for example, where four lines of address and a telephone number invariably follow the name.

But it pays to take greater care with anything more complex such as a customer records file. Here there's a fair likelihood the number of entries will differ from record to record, and some fields may even be left blank.

You can limit errors by splitting the required inputs into logical groups, if not singly. Thus you could still have a single prompt for a name and address, and then prompt singly for specific details thereafterwards.

Arrange for the prompts to come up singly, or in a different colour, or (at least) well spaced from the previous one. Clearing the screen after each screenful of prompts does seem much easier on the eye than one which simply scrolls ever upwards.

One other point here: with some programs an unusual input may call into use its own particular subroutine. In a datafile, for instance, additional items of information may be required for certain types of entry. Now if the user, through reasonable familiarity with the standard sequence of prompts, takes little notice of the new 'branch' of prompts—thinking nothing has changed—all sorts of errors could occur.

Error trapping routines are essential in all cases such as this—but it makes sense to warn the user in some way that the input requirements have changed. A simple flashing display or reversal of the prompt is usually quite sufficient. Or you could incorporate some sort of audible warning such as a beep, if your computer has this facility.

ERROR TRAPPING

The easiest way to prevent errors from being 'absorbed' by a program is to give the user the final option of accepting or rejecting what has already been entered. This can be done using a routine prompted by something like 'ARE ENTRIES CORRECT? Y/N'. Pressing N then simply restarts the input routine, whereas Y acts on the information present. But this is really necessary only on lengthy input routines.

If you decide to incorporate an entry acceptance routine within a program, combine all the answers in a single group if you can. This is easier to read and therefore much better than repeating the entire prompt and answer sequence on an individual basis.

But although entry acceptance routines provide one simple means of avoiding errors, programs have to be protected against incorrect entries.

For instance, can letters as well as numbers be entered where only letters or numbers are allowable? Always anticipate problems such as this when building up input routines in your own programs.

LENGTH LIMITS

In most datafile programs the length of input has to conform to the requirements of the program. A label program, for instance, must have its entries restricted to the physical limits of a label. After all, there's no point being able to enter an address line in excess of, say, 25 or 30 characters if there's no way entries of this length can be used because they will not fit on to the label.

You can use a prompt or suggest the real limit of entries by using indicators such as reverse characters or any other visual device to suggest the limits which are in effect.

Even so, additional programming must be provided to invalidate entries which are too long. In some cases, it may be preferable simply to truncate line entries at the correct length, and then rely on the entry acceptance routine to enter or reject this. But usually it's better to restart the input sequence at the point where the error occurred, providing a suitable error message such as 'ENTRY TOO LONG' followed by the prompt 'PLEASE RE-ENTER', or whatever you like.

One of the other main reasons for including entry length limits within a program such as a datafile is to conserve memory. After all, there's little point allowing 28 character spaces for a postcode entry if the maximum used is never more than eight. Trim back field lengths as far as possible to save memory and get the most from your datafiles.

INVALID VALUES

Setting limits is important for other reasons, particularly in programs which make use of numerical input for further calculations. For example, suppose the computer asks for three numbers (or calls in these numbers from memory), and divides the sum of the first two by the third. An earlier slip of a finger could enter a Ø rather than a 9 for the third. Or perhaps, as a result of another calculation within the computer, the value Ø is assigned to the third variable.

Unless the program is suitably protected the result is a 'division by zero' error message and an abrupt end to the program.

In a case such as this the protection needed is minimal as far as keyboard input is concerned—a simple keycheck routine can be used to restrict the range of acceptable values (see pages 284–288).

But for 'internal' calculations which could conceivably return a zero for use in further calculations specific error traps have to be used. Here it is a case of anticipating the worst and building into the program a value-checking routine using relational operators (see page 284). Something like this could be added:

```
IF A=Ø THEN GOTO 1ØØØØ
```

Line 1ØØØØ would then be the start of a routine that could either adjust the value to something other than Ø (but which was still acceptable to the program)—or advise the user that an invalid calculation was about to take place. In the latter instance the program could be redirected to a suitable input routine so an alternative value can be entered.

A particular range of values can be checked using something like the now familiar:

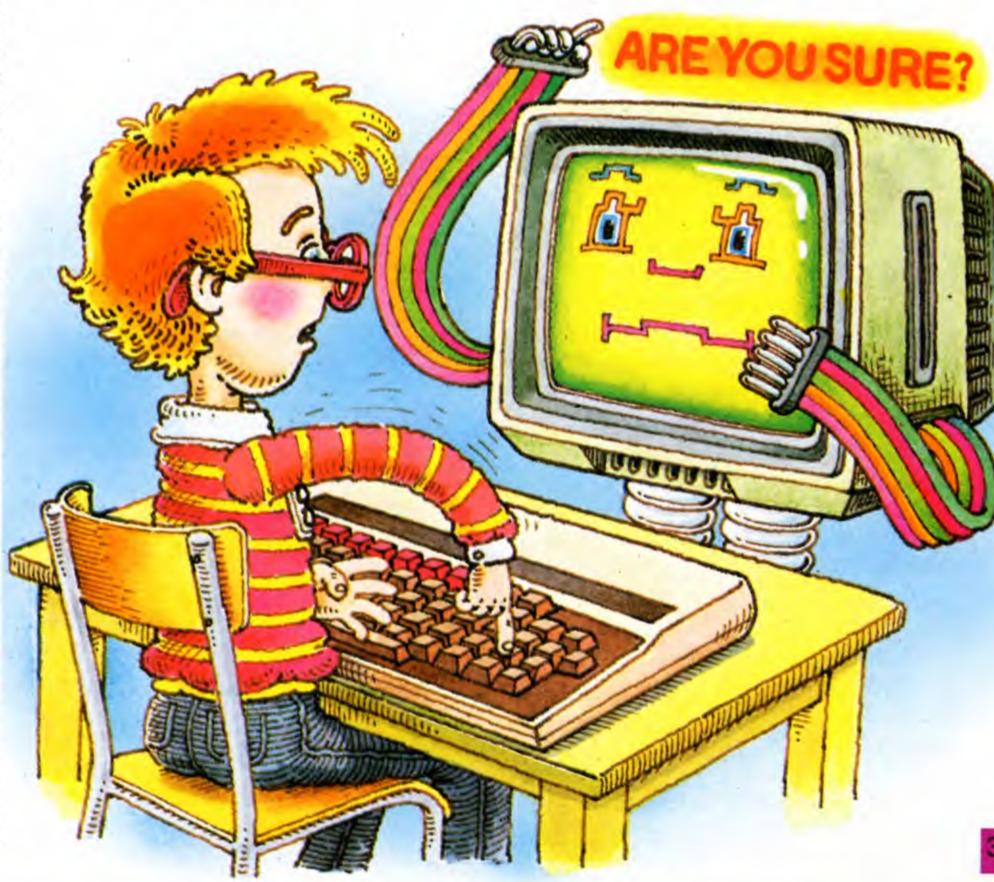
```
IF N>1ØØ OR N<1 THEN ....
```

This would typically return to the beginning of the input routine in the event of a value that is out of range.

In any case where there is a risk of an invalid input error under the direct control of the program user, use suitable prompts to suggest the available range of values and—if a mistake occurs—provide suitable error messages to let the user know why. These can be contained in a special subroutine accessed when, and if, necessary (see below).

CLEAR BUFFER

Certain computers—the Acorns and Commodores, for example—have to make use of a temporary storage area for keyboard/input entries. They do this using what is called a keyboard, or input, buffer. Errors can be





introduced if the wrong characters are entered by accident or earlier keypresses are still stored in the buffer.

For instance, in a program accepting a sequence of single key entries, pressing the key twice sends a second and probably unwanted character to the keyboard buffer ready for processing. The first input prompt accepts the first, and the second input prompt barely has time to display before being assigned the second keypress value and the program proceeds onwards.

To avoid situations such as this it is advisable to include this precautionary additional statement just before an input sequence:

```
POKE 198,Ø
```

```
*FX15,1
```

This completely clears the keyboard buffer prior to the expected input sequence which should follow immediately.

ERROR INDICATION

By providing the user with clear instructions on what to do when confronted with an input sequence, a great proportion of likely errors are eliminated at source. When errors do occur, however, clear indication of the problem can help to reduce the chances of repeating them.

What is needed here is a set of purpose-made error messages, and not the computer's system error messages. These can be defined and built into a program using a special subroutine used when a problem arises.

Typical uses of these are to indicate calculated or input figures which exceed the acceptable limits, duplication of names in a key field, incorrect input length and incorrect input type. In fact, imagine all the problems likely to occur in a program—and you can provide the error messages to go with them!

By using an array and variable arrangement you can define the number of error messages required and allow the program to adjust the value of the variable according to the errors.

The array for these error messages would be DIMensioned early in the program as part of the initialization procedure. This statement could be adjusted from time to time to include new potential errors that are discovered as the program is developed. So if you plan to use a set of nine program error messages, use a suitable DIM statement early in the program, typically like this:

```
1Ø DIM e$(9,2Ø): FOR z=1 TO 9: READ
   e$(z): NEXT z
2Ø DATA "Entry too long!", "Typing error!"
22 DATA "Wrong password!", "No data!"
24 DATA "Re-enter Data!", "Don't touch!"
26 DATA "Press (y)es or (n)o!",
   "Numbers only!","Letters only!"
```

```
1Ø DIM EM$(9):FOR Z=1 TO 9:
   READ EM$(Z):NEXT Z
2Ø DATA "ENTRY TOO LONG !",
   "TYPING ERROR !"
22 DATA "WRONG PASSWORD !",
   "NO DATA !"
24 DATA "RE-ENTER DATA !",
   "DON'T TOUCH !"
26 DATA "PRESS (Y)ES OR (N)O !",
   "NUMBERS ONLY !","LETTERS ONLY !"
```

Of course, you can define your own messages to suit the program conditions. Then, later on in the program, at each input point, the following entry check could be made:

```
1ØØØ LET a$="": LET em=Ø: INPUT a$
1Ø1Ø IF LEN a$>25 THEN LET em=1
```

Line 1Ø1Ø here is optional and could be replaced by a number of alternatives depending on the program. For example:

```
1Ø1Ø IF a$<>"credit" THEN LET em=3
1Ø1Ø IF a$="5" OR a$="9" THEN LET
   em=4
1Ø1Ø IF a$="" THEN LET em=5
1Ø1Ø IF a$="□" THEN LET em=6
1Ø1Ø IF a$<>"y" AND a$<>"n" THEN
   LET em=7
1Ø1Ø IF a$<"Ø" OR a$>"9" THEN LET
   em=8
1Ø1Ø IF a$<"a" OR a$>"z" THEN LET
   em=9
1Ø1Ø FOR z=1 TO LEN a$: IF a$(z)="O"
   THEN LET em=2
1Ø15 NEXT z
```

```
1ØØØ EM=Ø:INPUT A$
1Ø1Ø IF LEN(A$)>25 THEN EM=1
```

Line 1Ø1Ø here is optional and could be replaced with a number of alternatives depending upon the program. Some might be, for example: (Note that the fourth alternative does not work on the Commodores or Acorns as you cannot INPUT a space.)

```
1Ø1Ø IF A$<>"CREDIT" THEN EM=3
1Ø1Ø IF A$="5" OR A$="9" THEN EM=4
1Ø1Ø IF A$="" THEN EM=5
1Ø1Ø IF A$="□" THEN EM=6
1Ø1Ø IF A$<>"Y" AND A$<>"N" THEN
   EM=7
1Ø1Ø IF A$<"Ø" OR A$>"9" THEN EM=8
1Ø1Ø IF A$<"A" OR A$>"Z"
   THEN EM=9
1Ø1Ø FOR Z=1 TO LEN(A$):IF
   MID$(A$,Z,1)="O" THEN EM=2
1Ø15 NEXT Z
```

All the alternative keypress check lines here have the same line number as only one of these would be tied to a particular input routine. The exception is the last example which has to have a NEXT on a following line (1Ø15, here).

When a keypress test reveals an error, variable EM adopts a particular value which relates to a specific error message previously defined as part of array EM.

The first tests for entries greater than 25 characters, the second insists on the correct password. The third, typical of what might be used after a menu selection input, points out that no data is available if you choose options 5 and 9. Test four asks you to re-enter data if a null entry is made. The fifth responds cheekily if you happen to press the space bar. The sixth is a variation of the keypress test commonly seen after a yes/no prompt. The next two test that numbers only or letters only are entered, and the last tests to see that an O has not crept into the program in place of a Ø.

The program would then proceed, via a subroutine, to a message display routine which could take the form:

```
2ØØØ IF EM>Ø THEN PRINT TAB 6;e$(em)
```

```
2ØØØ IF EM>Ø THEN PRINT EM$(EM)
```

Typically the program would then return to the point where the incorrect input was made.

BOMBPROOFING

Earlier, we looked at how the usual system of keypress checks can be carried out to validate single key entries after, say, a menu. While this protects the program from incorrect entries, further refinements are necessary to provide complete protection for RUNning programs.

Imagine, for example, a program written for use by young children. Ideally, this would require them to press only one of perhaps two or three keys in order to progress through the program. All the other numeric and letter keys can be invalidated easily enough. But what about the control keys such as RUN/STOP and RESTORE, BREAK or ESCAPE? On the Spectrum you have to press two keys simultaneously to BREAK—virtually impossible to do by accident—but additional program safeguards are required by the other machines:

Several POKEs are available which can be used to disable certain of the more sensitive keys on the Commodores. For the 64, RUN/STOP can be disabled by using POKE 8Ø8,239. RUN/STOP and RESTORE can be disabled using POKE 8Ø8,251. Both are enabled using POKE 8Ø8,237.

For the Vic 20 RUN/STOP and RESTORE are disabled using POKE 8Ø8,128—and enabled using POKE 8Ø8,112.

On the Acorn computers, BREAK is the most critical of these, effectively resetting the computer when pressed. ESCAPE stops a program RUNning, displaying the escape message and the line number at the point where the program was interrupted.

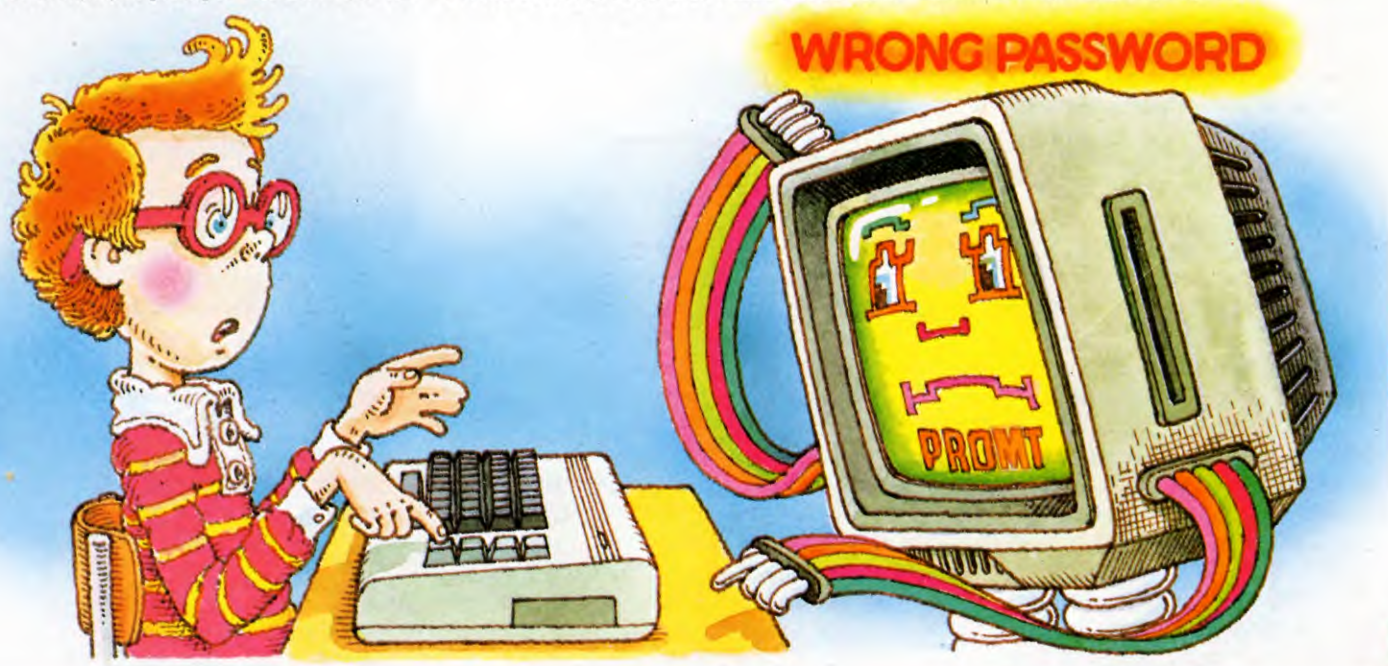
You would need to use a machine code routine to disable the BREAK key entirely but you can make it perform an OLD and RUN by using:

```
*KEY1ØOLD¦MRUN¦M
```

The OSBYTE call *FX229,1 can be used on all but the earliest BBC models to generate an ASCII code from the ESCAPE key instead of interrupting a BASIC program. *FX229,Ø restores the ESCAPE function.

Pressing the BREAK key on the Dragon keyboard stops a program RUNning but there is no easy way of disabling it without using the following POKE routine:

```
1Ø FOR K=415 TO 41Ø STEP-1: READ A:
   POKE K,A: NEXT K
2Ø DATA 228,237,4,2Ø3,228,236
```



SPECTRUM ASSEMBLER PROGRAM

If you find dealing with hex a bore, why not get your Spectrum to translate your assembly language listings into machine code for you? It'll even work out the jumps



Hand assembling can be a tedious business. Even if you know all the opcodes by heart and are familiar with the various addressing modes, translating a long assembly language program into machine code and then feeding it into your computer using your machine code monitor can be a laborious task.

But computers are particularly good at such exacting and repetitive work, so why not get your computer to do the translation for you? And the same program can be used to POKE the resulting machine code into memory at the same time.

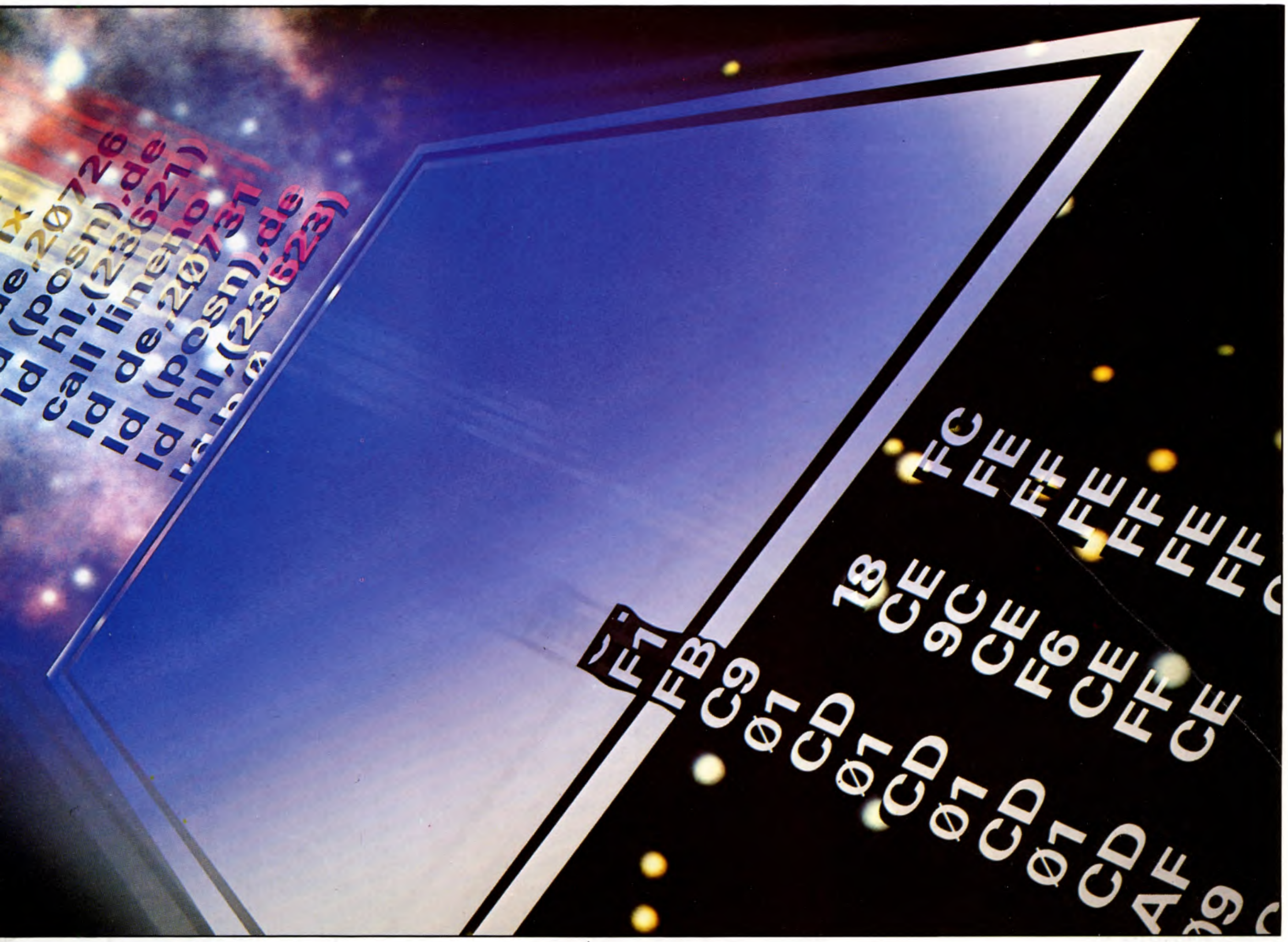
The following program is an assembler for the 48K Spectrum. Commodore 64, Dragon and Tandy versions follow in later chapters—the BBC Micro and the Electron do not need one, they have an assembler built in. And there is not enough room in memory to run similar assembler programs on the 16K Spectrum, the ZX81 or the unexpanded Vic 20. To make these programs suitable for publication they had to be written in BASIC, so they are not as fast as the machine code assemblers which are available commercially.

But they do work and will be able to assemble assembly language programs you will find in the books for your machine as well as the ones published in *INPUT*. You would be well advised to go and make yourself a cup of coffee while the Spectrum assembler is getting to grips with a long program, though. It may take some time.

THE ASSEMBLER

```
5000 DIM k$(110,4): DIM k(110):
   DIM m(110): LET h$=“0123456789
   ABCDEF”: LET b$=“”: LET
   g$=“0123456789abcdef”
5010 DIM t$(100,24): DIM r(100): DIM
   z$(100,6): DIM z(100)
5020 DIM b(9): LET b(1)=1: FOR i=2 TO 9:
   LET b(i)=b(i−1)+b(i−1): NEXT i
5030 DIM r$(8,4,4): FOR j=1 TO 4: FOR
   i=1 TO 8: READ r$(i,j): NEXT i: NEXT j
5040 DATA “Ø”,“1”,“2”,“3”,“4”,“5”,
   “6”,“7”,“nz”,“z”,“nc”,“c”,“po”,
   “pe”,“p”,“m”,“Ø”,“8”,“16”,
   “24”,“32”,“4Ø”,“48”,“56”,“hl”,
   “ix”,“iy”,“bc”,“de”,“hl”,“sp”,“□”
5050 DIM s$(8,2,4): DIM t(18): DIM
   u$(18,10): FOR j=1 TO 2: FOR i=1 TO
   8/j: READ s$(i,j): NEXT i: NEXT j: FOR j=1
   TO 18: READ t(j),u$(j): NEXT j
5060 DATA “b”,“c”,“d”,“e”,“h”,“l”,
   “(hl)”,“a”,“bc”,“de”,“hl”,“sp”,
   235,“de”,8,“af”,227,“(sp)”,
   6Ø742,“Ø”,6Ø758,“1”,6Ø766,“2”,
   233,“(hl)”,568Ø9,“(ix)”,65ØØ1,
   “(iy)”,1Ø,“(bc)”,26,“(de)”,
   6Ø767,“r”,2,“(bc)”,18,“(de)”,
   6Ø751,“r”,249,“sp”,6Ø743,“i”,
   6Ø759,“i”
5070 DEF FN b(x,i)=INT (x/b(i+1))
   −INT (x/b(i+2))*2
5080 DEF FN x(x,i)=x−b(i+2)*FN
   b(x,i)+b(i+1)
5090 DEF FN j(x,i)=INT x−b(i+1)*
   INT (x/b(i+1))
5100 DEF FN e(i$,j$)=(i$=j$( TO LEN (i$)))
5110 FOR i=1 TO 11Ø: READ k$(i),
   k(i),m(i): IF NOT FN e(“*”,k$(i))
   THEN NEXT i
5120 DATA “ld”,1Ø,1Ø,“ld”,26,1Ø,“ld”,
   6Ø767,1Ø,“ld”,6Ø759,1Ø,“ld”,2,138,
   “ld”,18,138,“ld”,6Ø751,138,“ld”,6Ø743,
   138,“ld”,64,22,“ld”,5Ø,2Ø2,“ld”,58,74,
   “ld”,249,139,“ld”,34,195,“ld”,42,67,
   “ld”,6Ø779,197,“ld”,6Ø771,199,
   “ld”,97,165,“ld”,64,54
5130 DATA “adc”,136,5Ø,“adc”,
   6Ø746,3,“add’,128,5Ø,“add”,9,
   149,“and”,16Ø,48,“or”,176,48,
   “xor”,168,48,“nop”,Ø,Ø,“sub”,
   144,48,“sbc”,152,5Ø,“sbc”,6Ø738,
   3,“cp”,184,48,“jp”,13Ø,45,“jp”,
   233,9,“jp”,568Ø9,9,“jp”,65ØØ1,
   9,“jp”,131,49,“jr”,96,45,“jr”, 88,41
5140 DATA “call”,132,45,“call”,
   141,41,“ret”,2Ø1,Ø,“djnz”,74,
   4Ø,“dec”,11,17,“dec”,5,16,“inc”,
   3,17,“inc”,4,16,“push”,197,17,
   “pop”,193,17,“di”,243,Ø,“ei”,
   251,Ø,“halt”,118,Ø,“ex”,235,139,
   “ex”,8,15,“ex”,227,143,“exx”, 217,Ø
5150 DATA “rst”,199,132,“rts”,
   192,5,“bit”,52Ø32,2Ø,“defb”,
   −256,4Ø,“ccf”,63,Ø,“scf”,55,Ø,
   “cpl”,47,Ø,“cpd”,6Ø841,Ø,“cpdr”,
   6Ø857,Ø,“cpi”,6Ø833,Ø,“cpir”,
   6Ø849,Ø,“daa”,39,Ø,“im”,6Ø742,
   8,“im”,6Ø758,8,“im”,6Ø766,8
5160 DATA “in”,6Ø736,13Ø,“in”,
   149,42,“ind”,6Ø848,Ø,“indr”,
   6Ø81Ø,Ø,“ini”,6Ø84Ø,Ø,“ldd”,
   6Ø84Ø,Ø,“lddr”,6Ø856,Ø,“ldi”,
   6Ø832,Ø,“ldir”,6Ø848,Ø,“neg”,
   6Ø74Ø,Ø,“otdr”,6Ø859,Ø,“otir”,
   6Ø851,Ø,“out”,6Ø737,2,“out”,
   141,17Ø,“outd”,6Ø843,Ø,“outi”,
   6Ø835,Ø
5170 DATA “res”,52Ø96,2Ø,“reti”,
   6Ø749,Ø,“retn”,6Ø741,Ø,“rl”,
   51984,64,“rla”,23,Ø,“rlc”,
```

- TRANSLATING ASSEMBLY LANGUAGE INTO MACHINE CODE
- WORKING OUT JUMPS AND BRANCHES
- COPING WITH LABELS
- ASSIGNING SPACE FOR DATA AND VARIABLES
- POKEING IN THE HEXADECIMAL



```
   51968,16,"rlca",7,Ø,"rld",6Ø783,
   Ø,"rr",51992,64,"rra",31,Ø,
   "rrc",51976,16,"rrca",15,Ø,
   "rrd",6Ø775,Ø
518Ø DATA "set",5216Ø,2Ø,"sla",
   52ØØØ,16,"sra",52ØØ8,16,"srl",
   52Ø24,16,"defw",-256,41
519Ø DATA "*",Ø,Ø: LET ii=i: LET k(11Ø)=ii
52ØØ LET b=Ø: LET ba=PEEK 23635+256*
   PEEK 23636+4: LET n=1
521Ø LET cc=1: IF PEEK ba< >234 THEN
   LET n=n-1: GOTO 525Ø
522Ø LET cc=cc+1: LET ba=ba+1: IF PEEK
   ba=13 THEN LET ba=ba+5:
   LET n=n+1: GOTO 521Ø
523Ø LET t$(n,cc)=CHR$ PEEK ba
524Ø GOTO 522Ø
525Ø FOR g=1 TO 1ØØ: LET r(g)=g-1:
   NEXT g: LET fh=1ØØ
53ØØ LET kØ=Ø: LET k9=99: LET pØ=Ø:
   LET vv=Ø
531Ø LET k=kØ: LET p=pØ
532Ø GOSUB 8ØØØ
533Ø GOSUB 7ØØØ: LET o$=i$: IF
   o$(1)="*" THEN PRINT o$;:
   GOTO 532Ø
534Ø IF o$="end" THEN PRINT
   '"□□□end last addr□";p-1
535Ø IF o$="end" THEN LET pØ=p:GOTO
537Ø IF o$< >"org" THEN GOTO
   54ØØ
538Ø GOSUB 7ØØØ: LET s=Ø: IF
   i$(1)="*" THEN LET s=p: LET i$=
   i$(2 TO )
539Ø LET p=VAL i$+s: PRINT
   '"□□□□org□";p;: GOTO 532Ø
54ØØ IF p=Ø THEN PRINT "(you forgot org)":
   LET p=5ØØØØ
541Ø LET p$=o$+"!": FOR i=1 +18*
   (o$< >"ld") TO 11Ø: IF o$< =k$(i)
   AND p$>k$(i) THEN GOTO 55ØØ
542Ø NEXT i: PRINT o$
```

```
543Ø IF i$(1)=“.” THEN LET i$=i$
    (2 TO)
544Ø GOSUB 9ØØØ:LET gg=r(g)
545Ø IF gg<=1ØØ THEN LET s=SGN
    z(gg): LET b=INT (ABS z(gg)/
    65536): LET r=ABS z(gg)−b*65536:
    LET q=PEEK r+256*PEEK (r+1):
    POKE r,FN j(p*s+q,8): PRINT
    “□poking□”;r;“□with□”;FN
    j(p*s+q,8): IF b THEN POKE
    (r+1),FN j((p*s+q)/256,8):
    PRINT “□poking□”;r+1;“□with□”;
    FN j((p*s+q)/256,8)
546Ø IF gg<=1ØØ THEN LET gh=r(gg): LET
    r(gg)=fh: LET fh=gg: LET gg=gh: GOTO
    545Ø
547Ø IF i$=“” THEN LET r(g)=p+1ØØ:
    GOTO 533Ø
548Ø PRINT “□(This line not recognised)”
549Ø GOTO 542Ø
55ØØ LET z=Ø: LET r=Ø: LET e=Ø: PRINT
    “□□□□”;o$;
551Ø LET op=k(i): IF m(i)=Ø THEN GOTO
    6Ø9Ø
552Ø GOSUB 7ØØØ: LET a$=i$: PRINT
    “□”;a$;
553Ø LET m=m(i): LET op=k(i): LET b=FN
    b(m,Ø): LET b7=b+2*FN b(m,7)+1: LET
    z=Ø: IF FN j(m,3)<2 THEN LET c$=a$:
    GOTO 572Ø
554Ø FOR j=1 TO LEN a$: IF a$(j)=“,”
    THEN GOTO 558Ø
555Ø NEXT j: IF o$=“rst” OR o$=“rts”
    THEN GOTO 558Ø
556Ø IF FN e(o$,k$(i+1)) THEN LET i=i+1:
    GOTO 553Ø
557Ø PRINT “□(two operands expected)”:
    GOTO 532Ø
558Ø LET b$=a$( TO j−1): LET
    c$=a$(j+1 TO )
559Ø IF FN b(m,2) THEN GOTO 565Ø
56ØØ IF FN b(m,7) THEN LET d$=c$: LET
    c$=b$: LET b$=d$
561Ø IF b$=“ahl”(b+1 TO b+b+1) THEN
    GOTO 572Ø
562Ø IF b$=“(c)” AND (o$=“in” OR
    o$=“out”) THEN GOTO 572Ø
563Ø IF (FN e(o$,k$(i+1))) AND (FN
    j(m(i+1),3)>=2) THEN LET i=i+1:
    GOTO 553Ø
564Ø PRINT “(first operand a or hl expected)”:
    GOTO 532Ø
565Ø IF FN b(m,1) THEN GOTO 569Ø
566Ø LET e$=(b$+“□□□”) ( TO 4):
    FOR j=1 TO 8: IF e$=r$(j,b7)
    THEN LET op=op+8*(j−1)*(b7<4)+
    16*(j−6)*(b7=4)*(j>3): LET z=
    (j−1)*(b7=4)*(j<=3): GOTO 571Ø
567Ø NEXT j: IF p$>k$(i+1) AND (FN
    j(m(i+1),3)>=2) THEN LET i=i+1:
    GOTO 553Ø
568Ø PRINT “(first operand bit or flag reqd)”:
    GOTO 532Ø
569Ø IF FN b(m,7) THEN LET d$=c$: LET
    c$=b$: LET b$=d$: GOTO 566Ø
57ØØ LET x=8: GOSUB 575Ø: IF e THEN
    GOTO 573Ø
571Ø IF c$=“” THEN GOTO 6Ø9Ø
572Ø LET x=1+15*b+7*(op<=6 AND
    op>=4 OR b$=“(c)”): LET b$=c$:
    GOSUB 575Ø: IF NOT e THEN GOTO 6Ø9Ø
573Ø IF e=2 OR p$>k$(i+1) AND FN
    j(m(i+1),3)=FN j(FN x(m,Ø),3) THEN LET
    e=Ø: LET i=i+1: GOTO 553Ø
574Ø GOTO 532Ø
575Ø LET r=Ø: IF FN b(m,4) AND
    FN e(“(”( TO NOT b),b$) THEN
    LET z2=FN e(“ix”,b$(2−b TO )+
    “□”)+2*FN e(“iy”,b$(2−b TO )+
    “□”): IF z2 THEN LET z=z2: LET
    e$=b$( TO LEN b$−NOT b): LET
    b$=“(hl)”(1+b TO 4−b): LET
    f$=“Ø”+e$(4−b TO )
576Ø IF FN b(m,3) THEN GOTO
    579Ø
577Ø LET e$=(b$+“□□□”) ( TO 4):
    FOR j=1 TO 8/(b+1): IF e$=s$(j,b+1)
    THEN LET op=op+(j−1)*x: RETURN
578Ø GOTO 581Ø
579Ø LET j2=9+9*(o$=“ld”): FOR
    j=j2−8 TO j2: IF k(i)<>t(j) THEN
    GOTO 581Ø
58ØØ IF FN e(b$,u$(j)) THEN RETURN
581Ø NEXT j: IF b$=“af” THEN IF FN
    e(“p”,o$) THEN LET op=op+48:
    RETURN
582Ø IF FN b(m,6) AND FN e(“(”,b$) THEN
    LET b$=b$(2 TO LEN b$−1): GOTO
    586Ø
583Ø IF FN b(m,5) THEN LET op=FN
    x(op+6*NOT b,6): GOTO 586Ø
584Ø IF p$>k$(i+1) THEN LET e=2:
    RETURN
585Ø PRINT “(cannot match operand to op)”:
    LET e=1: RETURN
586Ø LET r=65536
587Ø LET s=1
588Ø IF b$=“” THEN GOTO 6Ø8Ø
589Ø LET x$=b$(1): LET d$=b$(2 TO ): IF
    x$=“*” THEN LET r=r+p*s: LET
    b$=d$: GOTO 587Ø
59ØØ IF x$=“+” THEN LET b$=d$: GOTO
    588Ø
591Ø IF x$=“−” THEN LET b$=d$: LET
    s=−s: GOTO 588Ø
```





```
5920 IF x$="""" THEN LET r=r+CODE
    d$*s: LET b$=d$(2 TO ): GOTO 587Ø
593Ø LET q=Ø: IF x$<>"%" OR d$<"Ø"
    OR d$>="2" THEN GOTO 596Ø
594Ø IF d$>="Ø" AND d$<"2" THEN LET
    q=q*2+CODE d$-48: LET d$=d$(2
    TO ): GOTO 594Ø
595Ø LET r=r+q*s: LET b$=d$: GOTO 587Ø
596Ø IF x$<>"$" OR d$<"Ø" OR
    d$>="g" THEN GOTO 6ØØØ
597Ø LET x$=CHR$ CODE d$: FOR g=Ø TO
    15: IF x$<>h$(g+1) AND
    x$<>g$(g+1) THEN GOTO 599Ø
598Ø LET q=q*16+g: LET d$=d$(2 TO ):
    GOTO 597Ø
599Ø NEXT g: LET r=r+q*s: LET b$=d$:
    GOTO 587Ø
6ØØØ IF x$<"a" OR x$>"z" THEN GOTO
    6Ø4Ø
6Ø1Ø LET i$=b$: GOSUB 9ØØØ: IF i$<>""""
    THEN GOSUB 94ØØ
6Ø2Ø IF r(g)<>23ØØØ AND r(g)>1ØØ THEN
    LET r=r+(r(g)-1ØØ)*s: LET b$=i$:
    GOTO 587Ø
6Ø3Ø IF r(g)=23ØØØ OR r(g)<=1ØØ
    THEN LET gh=r(fh): LET r(fh)=
    r(g): LET r(g)=fh: LET fh=gh:
    LET z(r(g))=(p+SGN op+(ABS
    op>255)+2*(z>Ø)+65536*((b OR
    FN b(m,6)) AND o$<>"jr"))*s:
    LET b$=i$: GOTO 587Ø
6Ø4Ø IF x$<"Ø" OR x$>"9" THEN LET
    r=Ø: GOTO 6Ø7Ø
6Ø5Ø IF b$>="Ø" AND b$<":" THEN LET
    q=q*1Ø+CODE b$-48: LET b$=b$(2
    TO ): GOTO 6Ø5Ø
6Ø6Ø LET r=r+s*q: GOTO 587Ø
6Ø7Ø PRINT "(address not understood)"
6Ø8Ø LET r=r-(p+2)*(o$="djnz" OR
    o$="jr"): RETURN
6Ø9Ø PRINT TAB 16;: LET by=p/256: GOSUB
    619Ø: LET by=p: GOSUB 619Ø: GOSUB
    616Ø
61ØØ IF z THEN LET by=189+z*32: GOSUB
    618Ø: GOSUB 616Ø
611Ø IF op>=Ø THEN LET by=op/256:
    GOSUB 617Ø: GOSUB 615Ø: LET by=op:
    GOSUB 618Ø: GOSUB 615Ø
612Ø IF r=Ø THEN GOTO 532Ø
613Ø GOSUB 616Ø: LET by=r: GOSUB 618Ø:
    IF (b OR FN b(m,6)) AND o$<>"jr"
    THEN LET by=r/256: GOSUB 618Ø
614Ø GOTO 532Ø
615Ø IF z AND INT by AND NOT b THEN
    GOSUB 616Ø: LET by=VAL f$: GOSUB
    618Ø: LET z=Ø
616Ø PRINT "□";: RETURN
617Ø IF INT by<=Ø THEN RETURN
618Ø LET by=FN j(by,8): POKE p,by: LET
    p=p+1
619Ø LET by=FN j(by,8): PRINT h$(1+INT
    (by/16));h$(FN j(by,4)+1);
62ØØ RETURN
7ØØØ IF k>n THEN LET i$="end": RETURN
7Ø1Ø LET k1=k9+1: IF k9>=LEN t$(k)
    THEN LET i$="/missing/": RETURN
7Ø2Ø LET k9=k1: IF t$(k,k1)="□" THEN
    GOTO 7Ø1Ø
7Ø3Ø IF k9>LEN t$(k) THEN LET i$=t$(k)
    (k1 TO ): RETURN
7Ø4Ø IF t$(k,k9)<>"□" THEN LET
    k9=k9+1: GOTO 7Ø3Ø
7Ø5Ø LET i$=t$(k) (k1 TO k9-1): RETURN
8ØØØ IF k>Ø THEN IF t$(k) (k9 TO )>t$(99)
    THEN PRINT t$(k) (k9 TO );
8Ø1Ø POKE 23692,Ø: LET k=k+1: LET
    k9=Ø
8Ø2Ø PRINT : RETURN
9ØØØ LET x$=""""
9Ø1Ø IF i$<"a" OR i$>"z" THEN GOTO
    9Ø3Ø
9Ø2Ø LET x$=x$+i$(1): LET i$=i$(2 TO ):
    GOTO 9Ø1Ø
9Ø3Ø IF i$<>"""" THEN RETURN
94ØØ FOR g=1 TO vv: IF FN e(x$,z$(g))
    THEN RETURN
941Ø NEXT g: LET vv=vv+1: LET
    z$(vv)=x$: LET g=vv: LET r(g)=23ØØØ
942Ø RETURN
```



Tracing bugs in long programs

Keying in long programs like this one is very difficult to do without introducing bugs. The most common fault is to leave out a piece of DATA. If you get an 'OUT OF DATA' error message, check your DATA statements. Even a missing comma will cause problems. If your assembler does not work first time, don't despair. If you can't spot an obvious error, *INPUT* will be publishing a Spectrum trace to help you find the bugs.

The trace program prints on the screen the number of the line of BASIC being executed, as it is being executed. And *INPUT*'s trace gives the number of the statement being executed in that line as well. Using these facilities in conjunction with the published program, spotting errors is much easier.

HOW IT WORKS

You will notice that the assembler program starts on Line 5ØØØ. This is to leave room for your assembly language programs which must be entered as REM statements in BASIC lines.

Each assembly language instruction must be entered on a separate line, in a separate REM statement, in ordinary lowercase letters, not capitals.

At some point before you assemble the program you must CLEAR an area for your machine code to go into. The first line of your assembly language program should read something like:

```
1Ø REM org 32ØØØ
```

The 32ØØØ is the memory location where the machine code program will begin and it should, of course, be above the place you have CLEARed to.

If you forget to specify an **org**, or *origin*, the assembler will default to 5ØØØØ and it will put your program there.

Standard Z8Ø mnemonics are used with one exception, conditional returns. At the end of Z8Ø assembly language subroutines the mnemonic **ret** will return you to the main body of the machine code program. **ret** will work on this assembler.

But sometimes you want a conditional return, for example, **ret nz**. This means, return

if non-zero. With this assembler, you should use the syntax **rts nz** instead. And **rts** should be used instead of **ret** in any conditional returns, that is any return which has letters after the **ret**. For an unconditional return, that is an **ret** with nothing following, the **ret** is retained.

If hex numbers are used they must be preceded by a $ sign. Binary numbers need a % sign in front. And if there is no sign in front of a number the assembler will take it as decimal. Labels are any word that is not a command. Avoid the use of anything too similar and don't use numbers.

Assembly language programs must end with REM **end,** so the assembler knows when to stop.

Once you have keyed in your assembly language program all you have to do is RUN the program and your assembly language program—the *source code*—with its machine code equivalent—the *object code*—is listed on the screen. At the same time, the object code is POKEd into memory.

If at this point you feel that you may have made a mistake in one of the lines, you can LIST the program and edit the assembly language in the normal way.

Once all the machine code is assembled, the end address of the machine code routine is displayed.

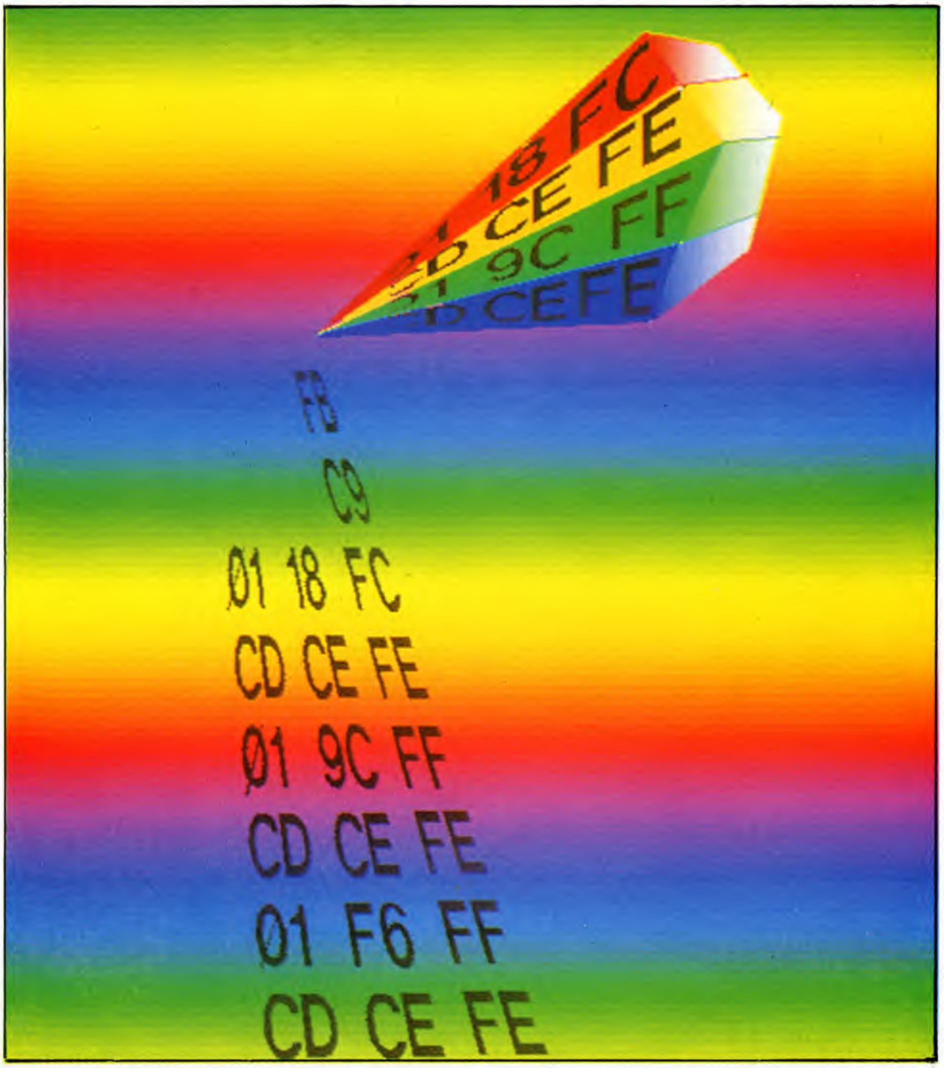
To execute the program you must use one of instructions used for running machine code programs (see page 282) like RANDOMIZE USR.

To SAVE the object code you should key in:

SAVE "name" CODE start address, no. of bytes

Name is the name you want to give to the routines. It must be in quotation marks. The keyword CODE tells the computer that it should save the program byte-by-byte, rather than a BASIC program that can be relocated anywhere in memory. Start address is the origin of the machine code program. And you can work out the number of bytes by subtracting the origin from the end address, and adding 1.

The assembler and the source code can be SAVEd using the normal SAVE routine (page 23).



Q+A

What happens if there is a bug in my assembly language program?

The assembler program given here has built-in error messages. The way that assembly language is structured means that the assembler can spot some of your errors. For example, some operations only work with the A or HL registers. If you try and perform them with any of the others, the assembler will tell you: 'first operand a or hl expected'.

If you have made a typing error and keyed in something that is not recognized as an operator, you will be told: 'This line is not recognized'. 'Two operands expected' means you have left out a vital piece of data after the command. 'Cannot match operand to op' means the data is of the wrong sort for the command. And 'first operand bit or flag required' means wrong data given with a branch or bit command.

TESTING

To test your assembler try keying in the assembly language right scrolling program given on page 323. Whether you hand assemble that program or feed it into your assembler, the machine code should read:

```
11 FF 57 21 FE 57 Ø6 CØ C5 1A Ø1 1F ØØ
ED B8 12 2B 1B C1 1Ø F3 C9
```

Note that the assembly language must be keyed in in ordinary lower case letters, not capitals otherwise the assembler will not recognise them. Now try the routine out and test that it works.

INPUT is not carrying a ZX81 assembler because it is not possible to write one in BASIC that could be used for the programs that will be covered in following chapters. If you are a ZX81 owner and are interested in machine code, it is suggested that you buy one of the machine code assemblers available commercially.

If you have one try it out on the right scrolling program on page 325. Whether you use an assembler or hand assemble that program you should get:

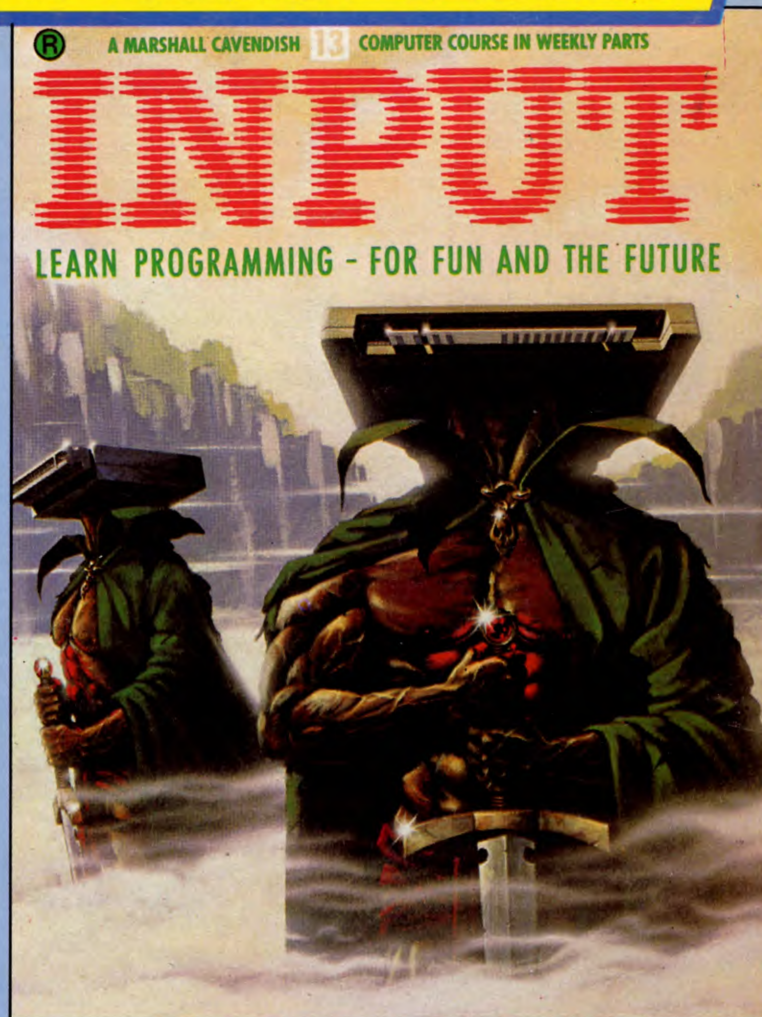
```
2A ØC 4Ø 11 16 Ø3 19 54 5D 13 Ø6 18 C5
1A Ø1 1F ØØ ED B8 12 2B 1B 1B C1 1Ø
F1 C9
```

CUMULATIVE INDEX

An interim index will be published each week. There will be a complete index in the last issue of *INPUT*.

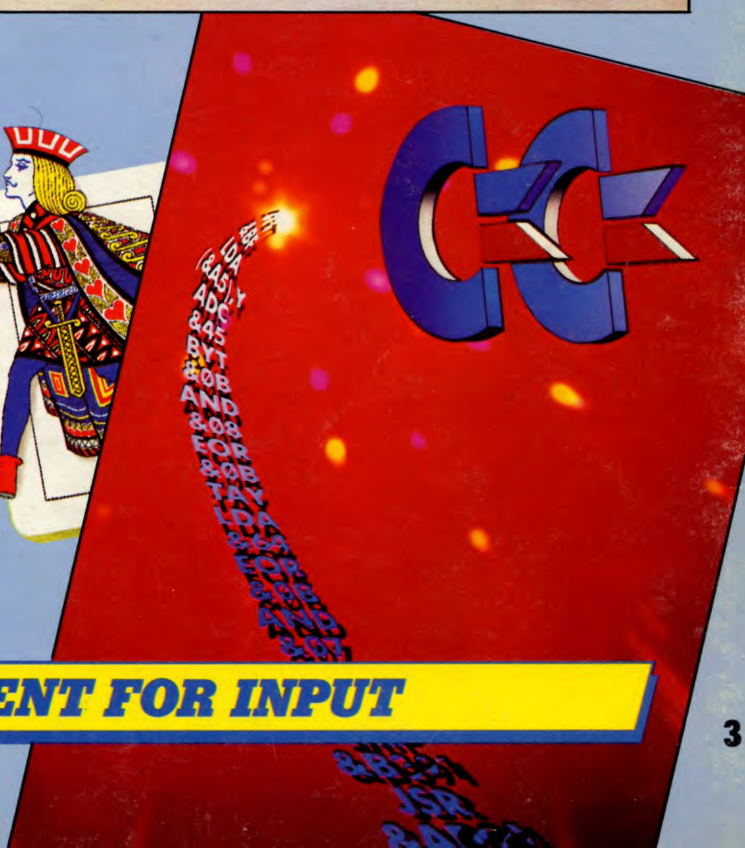
COMING IN ISSUE 13...

- *If you want to raise communication with the machine to a new level, what better than a SPEECH SYNTHESISER? Read this guide to the hardware for your computer*
- *Getting things in order is fundamental to everything from scoring games to compiling an index. Find out how to write different types of SORTING PROGRAMS*
- *Complete your ADVENTURE GAME by filling in the FINE DETAILS that give any game its own special character*
- *Explore the many uses of the POWER functions—for everything from area and volume to calculating the performance of a moving object*
- *Plus, for Commodore users, an ASSEMBLER PROGRAM to translate assembly language programs into machine code*









ASK YOUR NEWSAGENT FOR INPUT